ENABLING AND SUPPORTING THE DEBUGGING OF SOFTWARE FAILURES

A Thesis
Presented to
The Academic Faculty

by

James Alexander Clause

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 2011

ENABLING AND SUPPORTING THE DEBUGGING OF SOFTWARE FAILURES

Approved by:

Dr. Alessandro Orso, Advisor
College of Computing
Georgia Institute of Technology

Dr. Mary Jean Harrold
College of Computing
Georgia Institute of Technology

Dr. Spencer Rugaber
College of Computing
Georgia Institute of Technology

Dr. Andy Podgurski
Electrical Engineering & Computer Science Department
Case Western Reserve University

Dr. Milos Prvulovic
College of Computing
Georgia Institute of Technology

Date Approved: March 15, 2011

To Elizabeth,

for your love and support.

ACKNOWLEDGEMENTS

I would like to thank the many people whose help, support, and feedback helped to improve this dissertation.

First and foremost, I would like to thank my advisor, Dr. Alessandro Orso for his advice, support, and encouragement. I would also like to thank my committee members, Drs. Mary Jean Harrold, Spencer Rugaber, Andy Podgurski, and Milos Prvulovic, for all of their assistance and support.

I want to thank GJ Halfond and my other colleagues at Georgia Tech, both current and former, for the interesting conversations, ideas, and friendly competition they provided.

I would also like to thank Gregory Kapfhammer for encouraging me to pursue a degree in higher education, providing useful advice along the way, and for being an excellent role model on how to balance life, research, and teaching.

Finally, I would like to thank my wife, Elizabeth for her love, support, understanding, and willingness to delay life for a little longer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

SUMMARY

Debugging is an expensive activity that can be responsible for a significant part of the cost of software maintenance. This is especially true for today's software, whose complexity, configurability, portability, and dynamism exacerbate debugging challenges. Therefore, techniques for improving the efficiency and effectiveness of debugging can be extremely beneficial in reducing the overall cost of software development and, at the same time, improving software quality. In the last few years, there has been a great deal of research aimed at developing techniques that support automated or semi-automated debugging. These novel techniques have greatly contributed to the collective body of knowledge and have advanced the state of the art. However, an additional leap forward must be taken for automated debugging approaches to be effective when applied in real-life scenarios. Most current debugging approaches suffer from several limitations, including the assumption that developers are able to reliably and easily reproduce the targeted failure and that examining a faulty statement in isolation is always enough for a developer to detect, understand, and fix the corresponding bug.

This dissertation evaluates the following thesis statement: Program analysis techniques can enable and support the debugging of failures in widely-used applications by (1) capturing, replaying, and, as much as possible, anonymizing failing executions and (2) highlighting subsets of failure-inducing inputs that are likely to be helpful for debugging such failures. To investigate this thesis, I developed techniques for recording, minimizing, and replaying executions captured from users' machines, anonymizing execution recordings, and automatically identifying failure-relevant inputs. I then performed experiments to evaluate the techniques in realistic scenarios using real applications and real failures. The results of these experiments demonstrate that the techniques can reduce the cost and difficulty of debugging.

CHAPTER I

INTRODUCTION

Developers who want to eliminate a software fault must perform three main tasks. The first task is *fault localization*. It consists of identifying the program statement or statements responsible for the failure. The second task is *fault understanding*, which involves understanding the root cause of the failure. Finally, *fault correction* is the process of determining how to modify the code to remove the previously identified root causes. Fault localization, understanding, and correction are referred to collectively with the term *debugging*.[1]

An early study by Vessy demonstrated that debugging is an expensive activity that can be responsible for a significant part of the cost of software development and maintenance [85]. This cost has not decreased during the subsequent years. For example, 24,191 people were involved in the debugging of Windows Vista; an order of magnitude more than the approximately 2,000 who actually wrote the code [6, 31]. Looking forward, the cost of debugging is also unlikely to decrease because characteristics of modern software (e.g., complexity, configurability, and portability) exacerbate debugging challenges. Therefore, techniques for improving the efficiency and effectiveness of debugging are extremely beneficial in reducing the overall cost of software development and, at the same time, improving software quality. In the last few years, there has been a significant amount of research aimed at developing techniques that support automated or semi-automated debugging (e.g., [5, 20, 30, 42, 67, 97, 98, 100–102]). These techniques have greatly contributed to the collective body of knowledge and have advanced the state of the art. However, additional progress must be made for automated debugging approaches to be effective when applied in real-life scenarios. Today, most current debugging approaches suffer from one or both of the following limitations:

[1] Debugging can be divided into a larger number of finer-grained activities. However, for the work described in this dissertation, this level of detail is sufficient.

First, the majority of debugging techniques require that developers are able to reliably and easily reproduce the targeted failure. For these techniques, this is a reasonable and necessary assumption. However, there are many situations where it is difficult or even imposible for developers to recreate the failure in their development environment. This can be the case, for example, when a failure manifests on a machine that is not controlled by the developer (e.g., when a failure is discovered by third-party consultants or by end users) or when a failure involves sensitive information that developers are not allowed to access. The effects of not being able to easily reproduce a failure can range from, at best, an increase in the time and effort that is needed to debug the failure to, at worst, an inability to investigate and fix the failure.

Second, some approaches focus exclusively on reducing the number of statements developers need to examine when they investigate a failure. Although such approaches can work well for the simplest cases (e.g., isolated faults that involve a single statement), they are often inadequate when applied to more complex faults. For instance, faults of omission, which occur when part of a specification is not implemented, are notoriously problematic for debugging techniques that attempt to identify potentially faulty statements. The utility of these techniques is also limited when they are applied to long-running programs that process large amounts of information; failures in this type of program are typically difficult to understand without taking into account the data involved in the failures. In fact, existing approaches of this sort have been evaluated mostly on small subjects containing synthetic faults (e.g., [42, 67, 101, 102]), and there is little evidence that they would work in more complex cases.

The work presented in this dissertation addresses these limitations by providing algorithms and techniques that help developers reproduce difficult-to-reproduce failures and focus their limited resources on inputs that are relevant for their debugging tasks.

1.1 Thesis Statement

The overall goal of my research is to improve software quality. My dissertation work addresses this goal by developing new program analysis techniques that can help developers

debug program failures. My thesis statement is:

Program analysis techniques can enable and support the debugging of software failures by (1) capturing, replaying, and, as much as possible, anonymizing failing executions and (2) highlighting subsets of failure-inducing inputs that are likely to be helpful for debugging such failures.

To evaluate this thesis, I will develop techniques that implement the hypothesized capabilities and evaluate such techniques on real faults in medium- to large-sized applications.

1.2 Contributions

My dissertation research provides the following new techniques to the software engineering body of knowledge:

- ADDA—A technique that can enable and support the debugging of failures that are difficult to reproduce by capturing failing executions and providing developers with a minimized test case that recreates the captured failure.
- CAMOUFLAGE—A technique that can enable the debugging of failures that involve sensitive or privileged inputs by, as much as possible, automatically anonymizing such inputs.
- PENUMBRA—A technique that supports manual debugging efforts by automatically highlighting subsets of a program's inputs that are relevant for investigating a failure.
- Evaluations of the above techniques using real failures in widely-used applications that demonstrate that the techniques are feasible and useful.

Each of the techniques described above embodies one or more unique conceptual contributions.

The first technique enables the debugging of difficult to reproduce failures by recording, replaying, and minimizing failing executions. The conceptual contribution behind the recording and replaying portion is operating at the level of a program's environment. Previous techniques operated on low-level events such as memory reads. Moving away from such

low-level events allows my technique to minimize executions, something that is not possible with existing record / replay approaches. The contribution of the minimization portion is the definition and analysis of an algorithm for minimizing complete executions rather than individual inputs.

The contributions of the second technique are the concepts of path condition relaxation and breakable input conditions. These approaches are used to improve an existing anonymization technique and increase the likelihood that anonymized inputs can be sent to developers.

Finally, the contribution of the third technique is the use of dynamic tainting to identify subsets of a failure-inducing input that are likely to be useful for debugging a failure. This technique advances the state of the art because most existing debugging techniques focus primarily on reducing the amount of code that developers need to examine. In addition, when compared to techniques that do try to reduce the number of inputs a developer needs to examine, my approach provides comparable results but requires much less work on the part of developers.

1.3 Overview

Figure 1 presents a high level view of how the three techniques can work together to help reduce the cost of debugging software failures. First, ADDA's recording component is used to capture failing executions that occur on a remote machine. These failing executions are then minimized, using ADDA's minimization component, and anonymized, using Camouflage. After the captured failures are minimized and anonymized, they are sent back to the developer. The developer can then use the replay component of ADDA, to replay the captured failure, and Penumbra, to highlight subsets of the captured inputs that are relevant for investigating the failure. Note that while recording, minimizing, and anonymizing are typically performed on a remote machine, they also can work when used on the developer's machine. Note also, that while the techniques can work together, they can also be used independently. For example, Penumbra could easily be used to debug failures that were not captured with ADDA and Camouflage could be used with out minimizing the

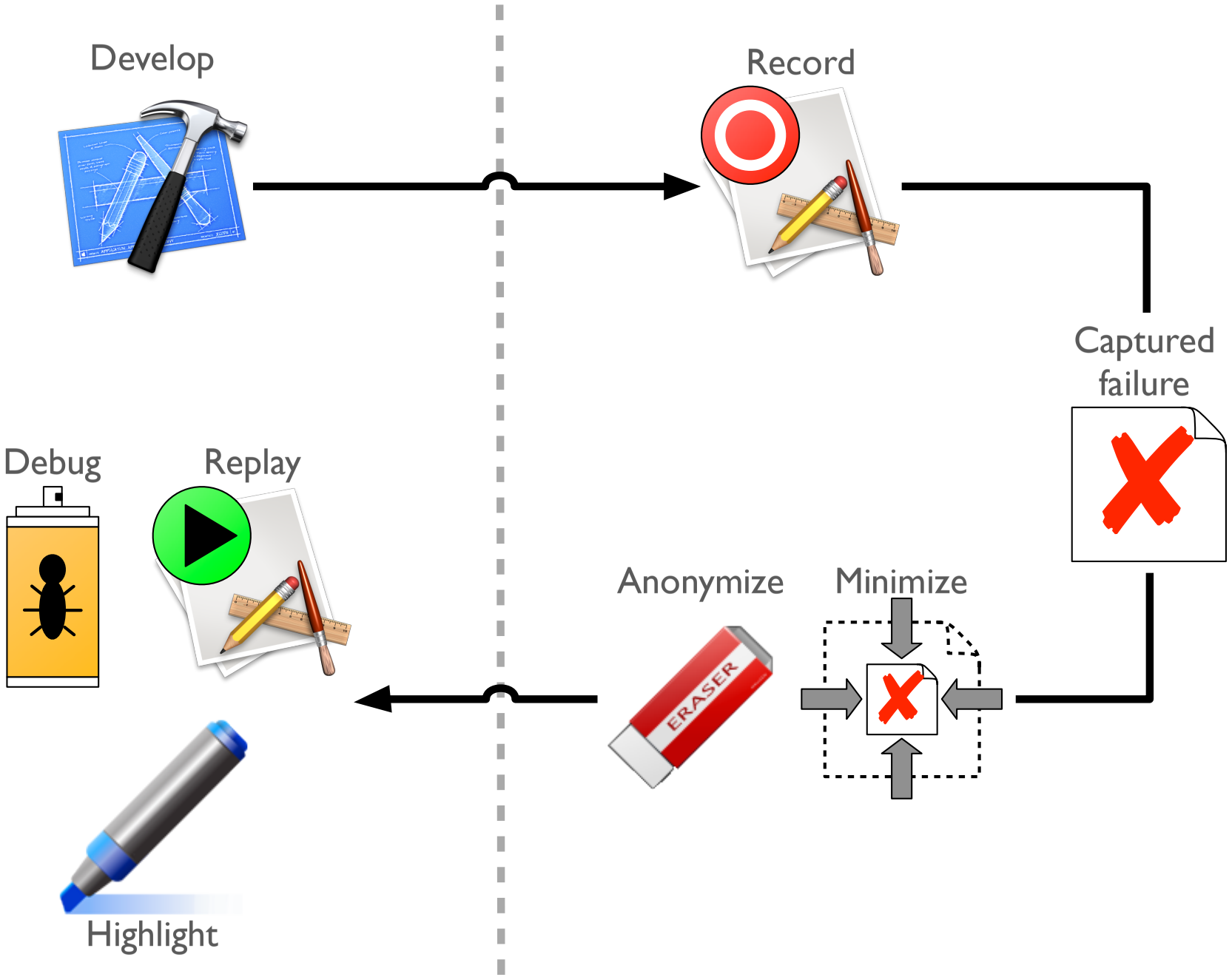

Figure 1: An overall view of how the techniques work together.

captured failures.

The following chapters are organized as follows. Chapter 2 provides background information on related work and definitions that are used throughout the text. Chapters 3, 4, and 5 provide detailed information about ADDA, CAMOUFLAGE, and PENUMBRA, respectively. ADDA is presented first because, as Figure 1 indicates, it is a prerequisite for CAMOUFLAGE. Finally, Chapter 6 concludes by discussing the merits of the dissertation and potential future work.

CHAPTER II

BACKGROUND

This chapter first defines terminology that will be used throughout the remainder of this work and then describes, at a high level, related work.

2.1 Definitions

The following definitions for "mistake," "fault," "error," and "failure" are taken from the "IEEE Standard Glossary of Software Engineering Terminology" [38]:

mistake: A human action that produces an incorrect result.

fault: An incorrect step, process, or data definition in a computer program.

error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

failure: The inability of a system or component to perform its required functions within specified performance requirements.

failure-inducing input: An input to a program that causes a failure to occur.

So in practice, a developer can make a *mistake* that results in a *fault* in a program. Such fault can then produce a *failure* in an execution when the program is run with a *failure-inducing input*. The difference between the program's expected and observed behaviors is the *error*.

When a failure is observed, a developer may attempt to fix the program by identifying and correcting the fault that caused the failure. I refer to this correction processess as *debugging*.

2.2 Related Work

Given the focus of this dissertation, there is a large amount of related work. To simplify the reader's task of comparing with my work, this section describes only related work in the general areas of debugging and leveraging captured information; a more detailed comparison with closely related work for each technique is provided within Chapters 3, 4 and 5.

2.2.1 General Debugging Techniques

As I mentioned previously, debugging is known to be a labor-intensive, time-consuming task that can be responsible for a large portion of software development and maintenance costs [79,85]. Common characteristics of modern software, such as increased configurability, larger code bases, and increased input sizes, introduce new challenges for debugging and exacerbate existing problems. In response, researchers have proposed many semi- and fully-automated techniques that attempt to reduce the cost of debugging.

In 1982, Weiser proposed one of the first techniques for supporting automated debugging and, in particular, fault localization: *program slicing* [91, 92]. Given a program P and a variable v used at a statement s in P, slicing computes all of the statements in P that may affect the value of v at s. If the value of v in s is erroneous, then the faulty statements that led to such erroneous value must be in the slice. Or, better, any statement that is not in the slice can be safely ignored during debugging. Slicing is therefore able to generate sets of relevant statements. However, in most realistic cases, these sets are too large to be useful for debugging.

To address this issue, researchers investigated variations of slicing aimed at reducing the size of the computed slices. Korel and Laski defined a technique, called *dynamic slicing*, that computes slices related to a particular execution, that is, a statement is in the slice only if it (1) may affect the value of v in s and (2) was executed in the specific execution considered. In subsequent years, different variations of dynamic slicing have been proposed in the context of debugging, such as critical slices [21], relevant slices [33], data-flow slices [102], and pruned slices [101]. These techniques can considerably reduce the size of slices, and thus better support debugging. However, the sets of relevant statements identified are often still

fairly large, and slicing-based debugging techniques are rarely used in practice.

A family of debugging techniques attempt to overcome the limitations of slicing-based approaches by following a different philosophy: identify potentially faulty code by observing the characteristics of failing program executions, often comparing them to the characteristics of passing executions. These approaches differ from one another in the type of information they use to characterize executions and statements—path profiles [68], counterexamples (e.g., [5, 30] and approaches based on model checking in general), statement coverage [42], and predicate values [49, 52]—and in the specific type of mining performed on such information. Additional work in this area has investigated the use of clustering techniques to eliminate redundant executions and facilitate fault localization [34, 41, 51, 67].

The majority of these techniques described in the proceeding paragraphs are code-centric in that they focus exclusively on one aspect of debugging—trying to identify the faulty statements responsible for a failure.

Although code-centric approaches can work well in some cases (e.g., for isolated faults that involve a single statement), they are often inadequate for more complex faults [13]. Faults of omission, for instance, where part of a specification has not been implemented, are notoriously problematic for debugging techniques that attempt to identify potentially faulty statements. The usefulness of code-centric techniques is also limited in the case of long-running programs and programs that process large amounts of information; failures in these types of programs are typically difficult to understand without considering the data involved in such failures.

To debug failures more effectively, it is necessary to provide developers with not only a relevant subset of statements, but also a relevant subset of inputs. There are only a few existing techniques that attempt to identify relevant inputs [12, 60, 100], with delta debugging [100] being the most known of these.

Delta debugging is an intuitive and effective technique that allows for reducing the data space of failing executions and for narrowing down the possible causes of such failures. However, current approaches based on delta debugging have two main problems. First, there are many cases in which identifying deltas in the inputs is difficult. Consider, for instance,

the common case of programs that receive inputs from different sources throughout their execution. For these programs, it is not clear what it means to divide the inputs into parts. More generally, in many cases domain knowledge must be incorporated in the delta debugging algorithm for it to be able to handle specific types of inputs. Second, delta debugging approaches rerun the applications being debugged multiple (in some cases many) times to observe the effect that partial inputs have on the execution and minimize program inputs based on that information. Frequent rerunning can be expensive and may cause the approach to be impractical (e.g., in the case of long running executions). Third, they also require complex oracles and setup, which can require a large amount of manual effort [8]. And fourth, they are only defined to operate on a single input. There have been a few attempts at addressing the problem of handling complex inputs (e.g., [60]), but no existing techniques target the remaining three issues.

The work described in this dissertation addresses the focus on code-centric approaches by presenting my technique for highlighting subsets of a failure-inducing input that are likely to be useful for debugging (see Section 5). This technique can complement code-centric debugging techniques because it focuses on identifying program inputs that are likely to be relevant for a given failure. It also overcomes some of the drawbacks of delta debugging because it needs a single execution to identify failure-relevant inputs and requires minimal manual effort. In addition, the minimization algorithm described in Section 3.2.4 demonstrates how delta debugging can be applied to multiple input simultaneously.

2.2.2 Testing and Debugging Techniques that Leverage Captured Data

The first project that proposed the use of field data to improve the way testing activities are performed is the Perpetual Testing project [65, 69, 95]. One of the main results of this project is the definition of *residual testing* [66], an approach for continuously monitoring test obligations not fulfilled in the development environment. The Expectation-Driven Event Monitoring project (EDEM) [36, 37] is also related, in that it proposes to collect program-usage data and compare them against an interaction model. Elbaum and Diep [26] investigate ways to efficiently collect field data (specifically, coverage data) that could be

used for improving the representativeness of test suites. These three projects are related to this research but address different aspects of testing and analysis, complementary to the work proposed. Residual testing focuses on monitoring coverage of deployed software. EDEM focuses on HCI aspects (i.e., usage and usability) by monitoring user actions on the GUI. The work by Elbaum and Diep is mostly focused on profiling field executions and defining efficient data collection techniques.

Another set of projects explicitly target debugging of software after deployment using field data and is therefore more closely related to this dissertation. The Microsoft Customer Experience Improvement Program (CEIP) records, on volunteering-users's machines, information on hardware and software configurations, performance and reliability, and program usage. The information is then used "to improve the products and features [users] use most often and to help solve problems" [57]. The Cooperative Bug Isolation project (CBI) [15, 48, 49, 83, 104] uses statistical techniques to sample and analyze execution data collected from real users to help debugging. (Because CBI's main goal is fault localization, this project is also mentioned in the previous section.) The Triage project [81] uses checkpointing and replay techniques combined with a set of predefined data analyses in an attempt to automatically collect information useful for the diagnosis of field failures.

CBI is a potentially effective debugging technique because it leverages field data not only to correlate regions of code with a particular failure, as most other fault localization approaches do, but also to provide some information, in the form of predicate values, on the local state of the program related to the failure. However, predicates provide only fragmented (and often noisy) information, and the current evaluations of the approach provide limited evidence of their general usefulness in the context of fault understanding and correction. Little information is available on how CEIP uses the collected information, but it appears that its goal is similar to CBI, that is, to support the maintenance effort and correlate problems with features or specific configurations. The Triage project has proven to be effective at providing information useful for diagnosing some types of failures. Its main limitation is that the failure analysis it performs on the user machines is predefined and performed in the same way for all types of failures. If the batch information provided

to the developers is insufficient for the investigation of the failure, the technique offers no options to the developer for refining the results (e.g., by collecting additional information). My technique for capturing, minimizing, and replaying failing executions (see Chapter 3) addresses the issue of only collecting partial information by allowing developers to reproduce capture failures. This allows them to use any debugging techniques that they desire rather than limiting themselves to only techniques based on the collected information.

An additional limitation of these techniques is their lack of concern regarding privacy and security. In many cases, these privacy and security concerns have prevented widespread adoption of many of these techniques and, because they rely on user participation, have ultimately limited their usefulness. Many of the earlier proposed techniques attempt to sidestep these concerns by collecting only limited amounts of information (e.g., stack traces and register dumps [3, 59] or sampled branch profiles [49, 50]) and providing a privacy policy that specifies how the information will be used (e.g., [2, 58]). Because the types of information collected by these techniques are unlikely to be sensitive, users are more likely to allow the techniques to be used. Moreover, because only a small amount of information is collected, it is feasible for users to manually inspect and anonymize such information before it is sent to developers.

Unfortunately, recent research has shown that the effectiveness of these techniques increases when they can leverage large amounts of detailed information (e.g., complete execution recordings [18, 87] or path profiles [15, 40]). Since more detailed information is bound to contain sensitive data, users will most likely be unwilling to let developers collect such information. In addition, collecting large amounts of information would make it infeasible for users to anonymize the collected information by hand. To address this problem, some of these techniques suggest using an input minimization approach (e.g., [96, 100]) to reduce the number of failure-inducing inputs and, hopefully, eliminate some sensitive information. Input-minimization techniques, however, were not designed to specifically reduce sensitive inputs, so they can only eliminate sensitive data by chance. In order for techniques that leverage captured field information to become widely adopted and achieve their full potential, new approaches for addressing privacy and security concerns must be developed.

One promising technique for addressing privacy concerns is the one proposed by Castro and colleagues [11]. The work described in Chapter 4 extends the work presented in [11] in several ways. From a conceptual standpoint, I present an improved approach that incorporates two novel techniques: *path condition relaxation* (see Section 4.3.2) and *breakable input conditions* (see Section 4.3.3). Intuitively, these techniques increase the strength of the general approach by decreasing the amount of information that is revealed about the original inputs. By doing so, they make it more difficult to reconstruct all, or even part, of the original input, which helps to ensure users' privacy.

CHAPTER III

ENABLING DEBUGGING BY CAPTURING, REPLAYING AND MINIMIZING FAILING EXECUTIONS

This chapter presents my technique for recording, minimizing, and replaying failing executions. The Chapter is organized as follows: Section 3.2 describes the technique in detail and provides a discussion of its correctness and worst-case performance characteristics; Section 3.3 describes the limitations of the technique; Section 3.5 describes the prototype implementation; and Section 3.6 discusses the evaluation of the technique.

3.1 Overview

The goal of the first technique is to support the debugging of failures by recording, minimizing, and replaying program executions. Existing approaches for record and replay have three main limitations with respect to this goal. The first two, efficiency problems and need for specialized hardware, are discussed in Section 3.7. The third limitation is that none of these approaches is amenable to minimization of the recorded executions. Most existing techniques record executions in terms of low-level events, such as memory accesses, and then reproduce these events exactly during replay. Recording executions at a low-level of detail allows for capturing all sources of non-determinism without having to worry about types and sources of inputs, and works well for replaying complete executions. When minimizing executions, however, the greatest challenge is to maintain consistency while eliminating parts of the execution. Low-level event logs contain events that are highly dependent on one another. Due to these dependencies, removing even a single event from a log is often enough to make the rest of the execution inconsistent and, thus, unusable. This issue has been experiences by myself and other researchers in previous work (e.g., [63, 71]).

The remainder of this Chapter discusses how I addressed the above issues to develop a record and replay technique that:

- is efficient enough to be used on deployed programs,
- does not require any specialized hardware,
- can replay executions in a sandbox, and
- is flexible enough to support minimization of executions.

3.2 The Technique

The technique works in three phases, intuitively illustrated by Figure 2. In the *recording phase*, while users run the software, the technique intercepts and logs the interactions between application and environment and records portions of the environment that are relevant to these interactions. If the execution terminates with a failure, the produced execution recording is stored for later investigation. In the *minimization phase*, using free cycles on the remote machines, the technique replays the recorded failing executions with the goal of automatically eliminating parts of the executions that are not relevant to reproducing the failure. Note that while minimized execution recordings reproduce the original failure, because the minimization process removes data, they do not necessarily reproduce the same execution states.

Finally, In the *replay and debugging phase*, developers can use the technique to replay the minimized failing executions and investigate the causes of the failures (e.g., within a debugger). Note that while the figure shows applications and minimized execution recordings being transmitted through the Internet, the technique works in the same manner if recordings are transmitted across a network or if the recording, minimizing, replaying, and debugging performed locally, on the same machine.

3.2.1 Execution Recording Format

One novelty of the approach is that it steps away from replay techniques that drive a program (almost) statement by statement based on a recorded event log. The approach uses an execution log when replaying entire executions but discard it almost completely otherwise. Intuitively, the technique treats executions as sequences of interactions between software and environment.

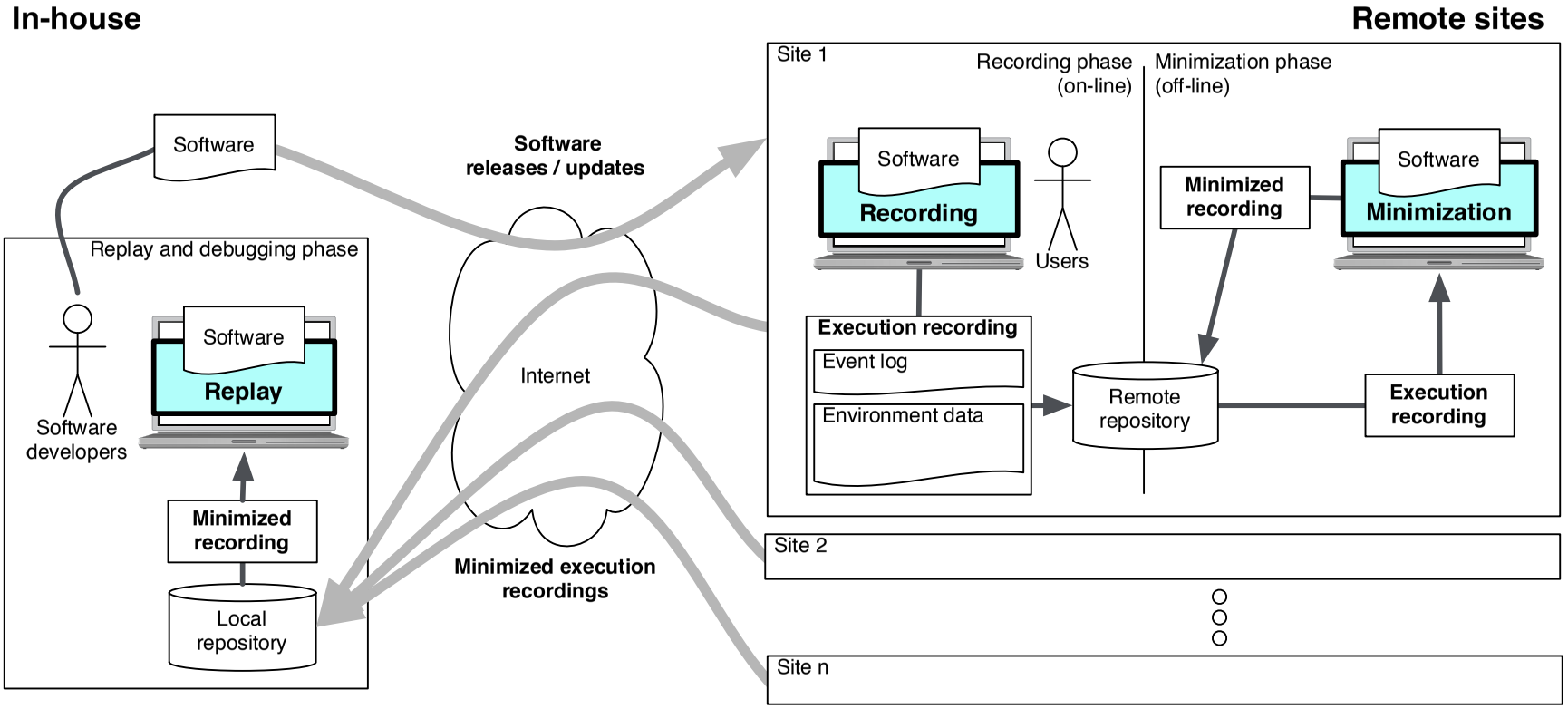

Figure 2: An intuitive scenario of usage of the technique.

While the software interacts with the environment through the Operating System (OS), by accessing different I/O streams (e.g., keyboard input, network, and files), the technique intercepts such interactions and produces an *execution recording* that consists of an event log and a set of environment data. The *event log* records relevant information about the observed interactions. The *environment data* consist of a dump of the input streams used by the software (*stream dumps*) and a copy of the files accessed by the software (*environment files*). Stream dumps also contain metadata that provide grouping and timing information for the data in the streams, used during minimization. Using the information in event logs and environment data, the technique can replay recorded executions in two ways. The first way is to perform a *complete replay* (see Section 3.2.3). In this case, the technique replays the execution as it was recorded by enforcing the sequence of events in the log and providing the program with data from the stream dumps at the appropriate times. The second way is to perform a *minimized replay* (see Section 3.2.4.2) by trying to eliminate as much data as possible from the stream dumps, discarding the event log, and letting the program run on this set of reduced inputs. In the case of a failing user execution, a minimized replay that still fails in the same way as the original execution would provide developers with a way to not only debug the captured failure, but also to do it on an execution in which some of the parts that are irrelevant for the failure have been eliminated.

3.2.2 Recording Executions

When recording executions, the technique intercepts three kinds of actions performed by the software: POLL, FILE, and PULL. Each action refers to a specific input stream (*current stream*, hereafter) or file. For each type of action, the technique updates the event log and the environment data, by either adding files to the set of environment files or modifying the stream dump for the current stream. The technique also logs any command line parameters for the executions and stores them as the first entry in the event log.

POLL actions are performed by the software to check for availability of data on an input stream. For each POLL action, the technique adds to the event log a *POLL event* with the following information: (1) a unique id for the current stream; (2) the outcome of

the POLL action (OK if data were available on the specified stream, or NOK otherwise); and (3) the amount of time elapsed before the software obtained a response to the POLL action. If the outcome of the POLL is positive, the technique also adds the amount of time elapsed, in the form of metadata, to the stream dump associated with the current stream.

FILE actions are interactions between the software and a filesystem. For each FILE action performed by the software, the technique checks to see if the action is the first reference to a specific file or directory.[1] If so, it copies the file into the set of environment data and stores the file's properties (e.g., owner, permissions, time stamps). The technique then increments a FILE-action sequence number and adds to the event log a *FILE event* that specifies (1) the current sequence number and (2) the name of the file being accessed. For subsequent accesses to the same file, the technique checks whether the file has been externally modified (by checking its modification time against the file's stored properties). If not, the technique increments the FILE-action sequence number, updates the file properties if needed, and does not log any event. Otherwise, a new version of the file is stored in the environment data and a new entry for the file is created in the event log. A numeric suffix distinguishes different versions of a file.

PULL actions are atomic reads of some amount of data from a stream. For each PULL action performed by the software, the technique first adds to the event log a *PULL event* that contains (1) a unique id for the current stream and (2) the amount of data being read in bytes. Then, the technique groups the data being read and appends the group to the appropriate stream dump, creating a new dump if one does not already exist.

Finally, if the last observed action accessed stream `s1`, and the new action either refers to a different stream or is a FILE action, the technique adds, as metadata, a separator marker to the stream dump for `s1`, in its current position. Separator markers delineate data read by separate PULL actions.

Figure 3 shows an example execution recording which consists of an event log (lines 1–10) and environment data for captured files and streams (lines 11–15 and 16). The first log

[1]For simplicity, in the rest of the paper, the term *file* to refers to both files and directories, unless otherwise indicated.

```
    Event log
 1.    PARAMS -I i
 2.    FILE 01 /etc/nsswitch.conf.1
 3.    FILE 02 /home/clause/.pinerc.1
       ...
 4.    POLL STREAM01 NOK 8000
 5.    POLL STREAM01 OK 5680
 6.    PULL STREAM01 1
 7.    POLL STREAM01 OK 1986
 8.    PULL STREAM01 3
 9.    FILE 41 /var/spool/mail/clause.2
10.    FILE 63 /var/spool/mail/clause.3

    Environment data: Files
11.    /etc/nsswitch.conf.1
12.    /home/clause/.pinerc.1
13.    /var/spool/mail/clause.1
14.    /var/spool/mail/clause.2
15.    /var/spool/mail/clause.3

    Environment data: STREAM01
16.    {time:5680}|m|{time:1986}|sko|{separator}...
```

Figure 3: Example of an execution recording.

entry at line 1 contains the command-line parameters used for the software. The following two entries (lines 2 and 3) are FILE events that indicate a first access to Version 1 of two different files. For each entry, there is a corresponding file in the set of environment files (restored during replay). The first POLL event in the log (line 4) is an unsuccessful polling of the input stream with id `STREAM01`. The third attribute of the event indicates that the POLL waited for eight second before receiving a negative response. Conversely, the next POLL event (line 5) received a positive response after 5,680 milliseconds, as shown both in the event log and in the metadata associated with the stream dump (shown within brackets and in boldface). After that, the program read one byte from stream `STREAM01` (line 6). The character that was read, in this case "`m`", is shown in the corresponding stream dump on line 16. The situation is analogous for the POLL and PULL entries at lines 7 and 8, except that three characters are read in this case, "`sko`", and the three characters appear thus as a group in the stream dump. Because the program then switched from an access to `STREAM01` to a FILE action (lines 8 and 9), the metadata for `STREAM01`'s dump contain a

separation marker in the current position. The two FILE events at line 9 and 10 correspond to the first accesses to Versions 2 and 3 of the user's mailbox, respectively. The difference in the value of the sequence numbers for the two events indicates that there have been 21 additional FILE actions between the two, but none of them was a first access to a new or externally-modified file.

3.2.3 Replaying Executions

The technique replays executions in a sandbox, which has two main advantages. First, it does not need the original user environment, which allows for replaying an execution in a different environment than the one where it was recorded. In particular, it allows for in-house replay of executions captured in the field. Second, it eliminates side effects on in-house machines such as deletion of files, modification of databases and transmission of files over the network caused by replayed executions.

This section describes how the technique performs complete replay. Minimized replay is described in the next section. To completely replay an execution, the technique sets the current position in the event log and in the stream dumps to their initial positions, initializes the sandbox that will contain the filesystem for the replay, and executes the software using the command-line parameters stored in the event log. Then, it lets the software run while intercepting the same software actions it intercepted during recording (plus output actions that need to be prevented from executing).

When the software performs a **POLL action**, the technique retrieves the outcome of the action and the response time from the current log entry. It then returns the retrieved outcome to the software after the specified response time. For the first POLL event in the example log, for instance, the technique would wait for eight seconds and then return a negative result to the software.

When the software performs a **FILE action**, the technique increments a FILE-action sequence number and then checks whether the current event in the log is a FILE event with a matching sequence number. If so, the technique retrieves the corresponding file version from the environment data and restores it, with the correct attributes, to the sandbox.

Otherwise, no file is restored. If the FILE action refers to the file by name (e.g., in a file open), the technique manipulates that parameter of the call so that it refers to the file in the sandbox. For the last FILE action in the example, the technique would copy `/var/spool/mail/clause.3` from the environment data, together with its attributes, to {`sandbox dir`}`/var/spool/mail/clause`.

When the software performs a **PULL action**, the technique reads from the event log the id of the stream being accessed and the amount of data to be read. It then reads that amount of data from the stream dump corresponding to the stream id and returns the data to the program. For the first and second PULL actions in the example, the technique would return one and three bytes, respectively, from the stream dump corresponding to id STREAM01.

3.2.4 Minimizing Executions

The goal of execution minimization is to transform failing execution recordings so they contain less data and, when replayed, reproduce the original failure in less time. Practically, this is accomplished by removing portions of the recordings that are not necessary for reproducing the failures. Algorithm 1 presents a pseudocode-level view of the minimization algorithm.

As input, the minimization algorithm takes (1) an execution recording which consists of the event log and captured files and streams as described in Section 3.2.2 and (2) a developer provided oracle. Because determining whether two executions exhibit the same failure requires the consideration of failure-specific data, it is difficult, if not impossible, to construct a general-purpose comparison mechanism. Instead, the minimization algorithm relies on the developer provided oracle to perform the necessary comparisons. More details about the construction of oracles is provided in Section 3.4.

At a high level, the algorithm removes unnecessary data from the given execution recording along two dimensions, time and data, in three steps, each of which is explained in detail below. Note that because data minimization replays the recorded execution multiple times, time minimization is performed first; Reducing the amount of time needed to replay an

Algorithm 1 Pseudocode describing how captured failing executions are minimized.

Input: $\langle L, F, S \rangle$: A captured failing execution where L is the event log, F is the set of recorded files, and S is the set of recorded streams.

Input: O: The oracle that was used to detect the captured failure. Note that the oracle must be capable of distinguishing among successful executions, executions that exhibit the same failure, and executions that exhibit a different failure.

Output: $\langle L', F', S' \rangle$: A minimized execution recording that when replayed reproduces the same failure as $\langle L, F, S \rangle$

```
1:  procedure MinimizeExecutionRecording(⟨L, F, S⟩, O)
2:      L_partial ← PartialFastForward(L)
3:      if CheckFailure(⟨L_partial, F, S⟩, O) then
4:          L_full ← FullFastForward(L_partial)
5:          if CheckFailure(⟨L_full, F, S⟩, O) then
6:              L' ← L_full
7:          else
8:              L' ← L_partial
9:          end if
10:     else
11:         L' ← L
12:     end if

13:     F' ← ∅
14:     for f ∈ F do
15:         if ¬CheckFailure(⟨L', F − {f}, S⟩, O) then
16:             f' ← MinimizeFile(f, ⟨L', F', S⟩, O)
17:             F' ← F' ∪ {f'}
18:         else
19:         end if
20:     end for

21:     S' ← ∅
22:     for s ∈ S do
23:         if ¬CheckFailure(⟨L', F', S − {s}⟩, O) then
24:             s' ← MinimizeStream(s, ⟨L', F', S'⟩, O)
25:             S' ← S' ∪ {s'}
26:         end if
27:     end for
28:     return ⟨L', F', S'⟩
29: end procedure
```

execution can save a considerable amount of wall-clock time during data minimization.

Step 1, which removes unnecessary time, corresponds to lines 2–12 in Algorithm 1. At line 2, the algorithm calls *PartialFastForward* which eliminates idle times due to unsuccessful POLL actions (i.e., polling calls that returned after a timeout because no data were available), by returning a copy of the passed event log where the timeout metadata field of all such POLLs is set to zero. The result of replaying such a recording is an execution in which reading data and processing inputs are replayed at their original speed, whereas the time during which the software was idle waiting for input is eliminated.

At line 3, the algorithm then calls *CheckFailure* which uses the provided oracle to check whether the execution produced by a modified recording, consisting of the partially fast forwarded event log and the unmodified files and streams, contains the same failure as the original execution. If the execution generated from the modified recording does not contain the same failure, the algorithm rolls back the modifications by restoring the original event log (line 11) and continues to Steps 2 and 3; no further time-minimization is attempted. However, if the resulting execution does contain the failure, the algorithm continues time-minimization.

FullFastForward, called at line 4, eliminates all idle time by setting the timeout metadata of all POLL events to zero. Executions replayed from recordings that have been fully fast forwarded contain virtually no idle time. After calling *FullFastForward*, the algorithm checks whether an execution recording consisting of the fully fast forwarded log and the unmodified files and streams still produces the original failure (line 5). Similarly to the previous check, if the resulting execution does not contain the failure the algorithm rolls back the changes (line 11) and if the execution does contain the failure, the algorithm retains the changes and continues to Steps 2 and 3.

At a high level, Step 2 (lines 13–20) constructs a copy of the stored files (F') that only contains necessary data. To accomplish this, the algorithm iterates over each of the stored files (line 14) and first tries to completely remove the considered file f. If an execution recording without f still generates an execution that contains the desired failure, f is unnecessary and is not added to F' (line 15). In the empirical evaluation (see Section 3.6),

a large portion of files (around 86% on average) were completely removed by this step. If f is necessary, however, further minimization is attempted (line 16) by calling *MinimizeFile*.

MinimizeFile is designed as an extension point for the algorithm. The rationale for this decision is to enable the use of developer-provided minimization algorithms, which can take into account the structure of the files used by the software and be more effective in reducing their size. Customized minimization algorithms must satisfy two requirements. First, for correctness, they must use the provided oracle to determine whether an execution recording that incorporates their changes generates an execution with the desired failure. And second, they must return a modified copy of f which is then added to F' (line 17). If no customized algorithm is provided, the algorithm defaults to using the delta debugging algorithm [100].

Step 3 is identical to Step 2 except that it minimizes recorded streams instead of files. It tries to completely remove each individual stream (line 23) and, if it can not, it calls *MinimizeStream* (line 24). Like *MinimizeFile*, *MinimizeStream* is a developer provided algorithm that removes unnecessary parts of the passed stream. It must satisfy the same requirements as *MinimizeFile* and also defaults to delta debugging if a custom algorithm is not provided.

As a possible extension, Steps 1 through 3 can be performed several times, until no additional minimization is achieved or until a given time threshold is reached. Additional iterations might expose possible synergies between different steps (e.g., eliminating parts of a stream dump may allow for eliminating a file that was previously necessary).

3.2.4.1 Analysis of the Minimization Algorithm

This section discusses the worst-case runtime and space usage of the of the minimization algorithm shown in Algorithm 1 and provides a proof of it's correctness.

Worst-case performance characteristics. The worst-case performance of the algorithm occurs when no minimization is possible. In this case, the algorithm will need enough space to store a complete copy of the original execution recording (i.e., $L' = L$, $F' = F$, and $S' = S$).

When considering runtime complexity, the call to *CheckFailure* is the most expensive operation since it replays the execution recording. Step 1 of the algorithm (lines 2–12) uses a constant number of calls to *CheckFailure* (i.e., 2) while Steps 2 and 3 call *CheckFailure* once for each $f \in F$ and $s \in S$. Thus far, in the worst-case, the algorithm requires $2 + |F| + |S|$ calls to *CheckFailure*.

However, *MinimizeFile* and *MinimizeStream* will also call *CheckFailure* to validate their modifications. Because these functions can be overridden it is impossible to analyze all possibilities. Therefore, I assume that the default algorithm, delta debugging, is used.

According to Zeller and Hildebrandt [100], the worst case for delta debugging also occurs when no minimization is possible. In this circumstance, delta debugging requires $|c|^2 + 3|c|$ tests (i.e., calls to *CheckFailure*) to minimize an input where c is the set of possible changes that the algorithm will apply to the input.

In Algorithm 1, delta debugging is run on each captured file (the call to *MinimizeFile* at line 16) and stream (the call to *MinimizeStream* at line 17). For ease of explanation, I will refer to the file or stream that is being minimized as e and the set of all files and streams as E. When minimizing $e \in E$, delta debugging considers the removal of each individual element (e.g., character) in e as a possible change (i.e., $|c| = |e|$ where $|e|$ is the size of the file or stream being minimized). Therefore minimizing every file and stream requires at most $\sum_{e \in E} |e|^2 + 3|e|$ calls to *CheckFailure*.

In the worst-case, Algorithm 1 requires $2+|F|+|S|+\sum_{e \in E} |e|^2+3|e|$ calls to *CheckFailure*. In short, it is polynomial in the size of the captured files and streams.

Correctness. Algorithm 1 is correct if:

For all execution recordings R' produced by minimizing an execution record R:

(1) R' and R reproduce the same failure

and

(2) the size of R' is less than or equal to R.

As long as the provided oracle meets the necessary prerequisites (i.e., that it can accurately detect the presence or absence of the failure captured in the original execution

recording), *MinimizeExecutionRecording* satisfies the definition of correctness.

The first condition, that the minimized and original execution recordings produce the same failure, holds because each change performed by the algorithm is checked (lines 3, 5, 15, and 23) and reverted if the resulting execution recording no longer produces the failure. The second property, that the size of the minimized recording is no larger than the original, is also satisfied. Consider a worst case situation, where every part of an execution recording is necessary for reproducing the captured failure. Under these circumstances, the components of the modified execution recording L', F', and S' will be the same as the original components L, F, and S. Because the algorithm never adds any data to the modified execution recording that was not taken from the original, the modified execution recording will never be larger than the original.

If delta debugging is used for *MinimizeFile* and *MinimizeStream* then R' is also guaranteed to be *1-minimal* [100]. That is, removing any single element from R' will prevent it from reproducing the original failure. Note that being 1-minimal does not guarantee that R' can not be minimized further. For example, assume that f1 and f2 (1) are elements in a minimized execution recording, (2) are not necessary for reproducing a failure and (3) have a circular dependency. For this execution recording, no further minimization can be done by the algorithm because it is not possible to remove f1 independently of f2 or vise versa, but it is possible to remove them both at the same time.

3.2.4.2 Replaying Minimized Executions.

The discussion so far has purposely glossed over technical issues related to the replay of minimized executions. Whereas time-based minimization is still driven by the event log, and thus ensures some degree of control on the execution of the software, input-based minimizations do not rely on the event log. The reason for this difference is that input-based minimizations reduce or even eliminate inputs (files and streams), which can result in a potentially large number of log entries being inconsistent with the actual replayed execution. For example, consider the effect of removing all data from stream `STREAM01` in the log of Figure 3. All successful POLL events for that stream would be inconsistent with

the environment, and all PULL events for the stream could not be replayed because there would be no data to read.

This simple example is representative of what would happen with traditional record and replay techniques that are completely based on recording and replaying a log. The technique uses a log when performing complete (and fast-forward) replay, but can also replay without having to enforce a specific sequence of events. Thanks to its notion of environment and to the way it records files and streams, the technique can replay minimized execution recordings by (1) still intercepting the usual software actions, and (2) responding to these actions based on the state of the minimized environment data instead of the information in the log, as follows.

If the software performs a **POLL action**, the technique returns a negative result if there is no data in the corresponding stream dump. Otherwise, it returns a positive response either right away (complete fast forward mode) or after the timeout specified at the current position in the stream dump (partial fast forward mode). If the software performs a **PULL action**, the technique reads and returns the next group of data in a stream, if any. The situation for a **FILE action** is slightly more complex in the presence of files with multiple versions. In such a situation, without relying on sequence numbers, the technique cannot distinguish between actions that should access a new version of a file and actions that should access the current version. To address this issue, the technique uses a single version of a file throughout a (minimized) execution. In particular, it tries in turn the last, the first, and then the intermediate versions.

Obviously, there are many ways in which an execution could take a wrong path and either not fail or manifest a different failure. However, as the empirical results show, the flexibility of the approach allows for identifying many executions that maintain their behavior of interest (the failure) even when run on a different set of environment data. Section 3.6.2.3 reports some data on the effectiveness of the different parts of the minimization.

3.3 Limitations of the Technique

In theory, the technique is able to record and replay arbitrary executions. However, there are practicality issues that can limit the number of situations in which the technique can be applied. The available space for storing recordings is one such issue. For example, for some types of applications such as databases, storing copies of the data accessed by the application (e.g., the entire database) is infeasible. The available space also places limits on the length of executions that can be captured. In my evaluation, I observed recording rates of several megabytes of data captured per minute of execution. While the specific ratio of captured data per unit of execution time can vary widely depending on the application that is being recorded and its workload, it is unlikely that a typical user's computer would have enough storage to save recordings for executions lasting several days or weeks.

In addition to the issue of storage, there may also be potential issues with the amount of runtime overhead that is imposed by the technique. Although the technique only imposes a minimal amount of runtime overhead when recording (see Section 3.6.2.2), there may be some types of time-sensitive applications, for example web or other types of servers, where even a small amount of overhead is unacceptable.

Even with these practical limitations, the technique is still useful. Many types of commonly used software (e.g., web browsers, word processors, spread sheet applications, and email and chat clients) are used in a manner that is congruous with the technique's prerequisites. To demonstrate the applicability of the technique, I recorded my usage of the latest versions of four common desktop applications: `Firefox`, a commonly used web browser; `Aquamacs`, a text editor; `iChat`, a chat client; and `Mail`, Apple's email program. The week that I monitored was typical of my standard usage: Firefox was to browse the Internet and watch online videos; `Aquamacs` was used to write and edit documents, primarily this disseratation; `iChat` was used to converse with friends and family, and `Mail` was used to send and receive my work email.

Table 1 presents the average amount of data read, per day, by these applications over the monitored week. In the table, columns 2–4 show the amount of data read from the network, file system, and user input (i.e., keyboard and mouse), respectively. For network

Table 1: Average amount of data read per day over a one week period.

Application	Network (MB)	File system (MB)	User input (MB)
Firefox	343	5	0.08
Aquamacs	—	51	0.2
iChat	2	2	0.007
Mail	28	91	0.04

and file system I intercepted the relevant system calls and recorded the amount of data that was read. For the mouse, I recorded the screen coordinates of the mouse during click presses and releases. For replay purposes, vertical and horizontal movement can be reconstructed from click information so there was to need to record it as well.

Overall, the data distribution for each application is unsurprising and conforms with the applications' intended uses (e.g., `Firefox` reads most of its data from the network while `Aquamacs` reads most of its data from the file system). One commonality across all of the considered applications is that the amount of data read from the network and the file system is much greater than the amount coming from user input.

Although this data is anecdotal, it does provide a preliminary indication that recording the executions of some common desktop applications is feasible.

3.4 Constructing Oracles

This section provides guidance for developers for constructing oracles to be used with the capture / replay and minimization techniques. This guidance is based on both my personal experience and my observations of other Georgia Tech Ph.D. students in writing oracles for several types of failures in real programs.

At a high level, an oracle is responsible for examining an execution and determining which of three outcomes has occurred: (1) the execution completes without exhibiting a failure, (2) the execution exhibits the considered failure, or (3) the execution exhibits some other failure.

When constructing oracles there are several important factors that need to be considered. The first thing that should be identified are the properties that can uniquely identify the failure. These are the properties that, if found, will indicate that the considered failure has

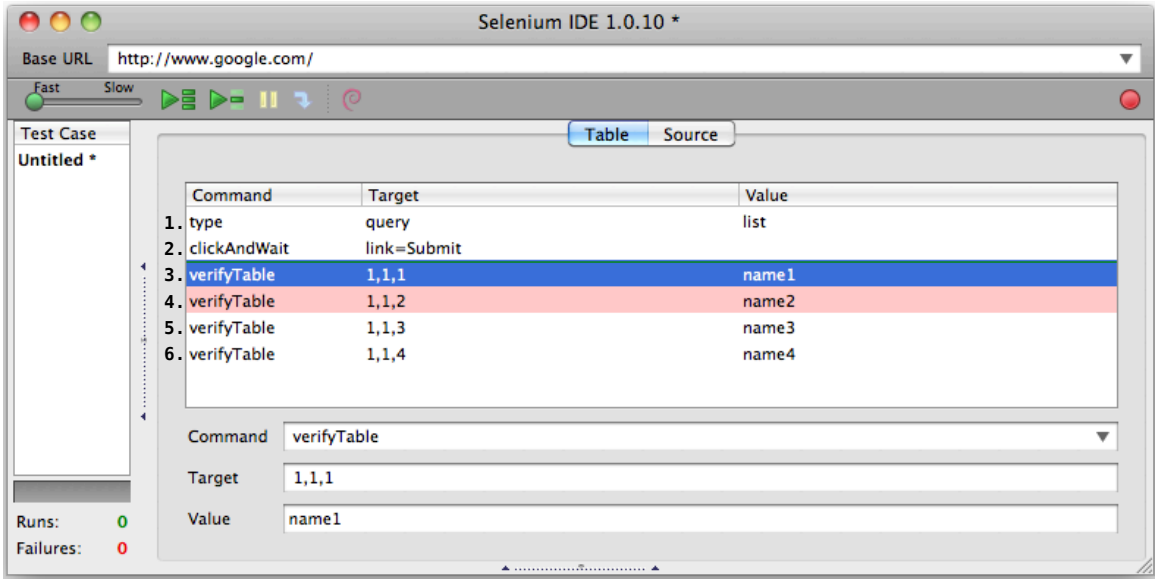

Figure 4: Example oracle for a simple web application.

occurred.

The simplest type of failure property, and the type most often seen in the literature, is one that examines a program execution for a specific output (e.g., whether the execution terminates abnormally or if a specific string is written to the console). Depending on the failure under investigation, solely checking the output can uniquely identify a failure. For example, the oracle for the motivating example provided by Zeller and Hildbrandt in [100] simply checks whether or not Mozilla terminates with a segmentation fault. Similarly, the oracle provided for the failure in `gcc` in the delta debugging tutorial checks whether the string "fatal signal 11" is printed to the standard error stream [99].

The output of command line applications and other programs that write to standard streams can be easily inspected by using a tool such as `expect` or the stream redirection and string search capabilities of many Unix shells or general purpose programming languages. Inspecting the output of graphical programs is slightly more challenging, but it can be accomplished by leveraging GUI testing tools (e.g., Selenium [72] for web applications and Eggplant [80] or Test Anywhere [4] for desktop applications). Figure 4 shows an oracle, written using Selenium, for a simple web application.

The main part of the oracle is shown in the table at the center of the figure. Each line in

this table is a Selenium script command. The "type" command at line 1, tells the underlying web browser to put the word in the Value column (i.e., "list") into the input field described by the Target column (i.e., the "query" field). The "clickAndWait" command at line 2 clicks on the hyperlink described by the Target column (i.e., the hyperlink with "Submit" as the link text) and then waits for the new page to finish loading. The "verifyTable" commands at lines 3–6 are asserts that check whether the contents of the first table on the new page are the same as the values in the Value column. Specifically, this example checks that the value in column one, row one is equal to "name1", the value in column one, row two is equal to "name2", etc. Although this example is simple, it illustrates the necessary concepts for writing oracles that test the GUI components of applications.

Currently, there is no easy mechanism for automatically inspecting other types of output such as audio or physical objects (e.g., printed sheets of paper). However, these types of output are significantly rarer and are less likely to be needed in practice.

Although checking a program's output can be sufficient, it is inadequate for some types of failures. For example, failures that do not crash the execution or failures that produce output that is not unique to the failure. The failures in `pine` described in Section 3.6.1 fall in this category. `pine` has an internal error handling that detects unexpected behaviors. Because all failures are routed through this routine, it is possible that different failures may produce the same output and consequently can not be uniquely identified by a simple oracle.

To handle these more difficult to identify failures, a useful strategy is to examine the internal state of the execution using some type of program introspection. In practice, the most common introspection tools are debuggers (e.g., `gdb` for compiled applications, `jdb` for Java, etc.).

Figure 5 shows an example oracle for one of the failures in `pine`. This version is written in `Python`, but similar concepts could be applied in any general purpose language. Line 2 runs `pine` and checks whether the execution terminates with an exception. If it does not, line 13 is executed and the oracles return "PASS" indicating that no failure occurred. If the execution does terminate with a failure, then lines 5–12 are executed. Line 5 invokes `gdb`

```
 1. def oracle():
 2. try:
 3.   subprocess.check_call([PINE, "-f", MAILBOX_NAME, "-Ii,q,y"])
 4. except subprocess.CalledProcessError, e:

 5.   p = subprocess.Popen(["gdb", "-q", PINE, "core", "-x", "cmds"],
                           stdout=subprocess.PIPE,
                           stderr=subprocess.STDOUT)
 6.   p.wait()

 7.   out = p.stdout.readlines()

 8.   bt = [line.split()[3] for line in out if line.startswith("#")]

 9.   if bt == BACKTRACE:
10.     return self.FAIL
11.   else:
12.     return self.UNRESOLVED

13. return self.PASS
```

Figure 5: Example oracle for a failure in `pine`.

on the generated core dump and runs a sequence of commands that print the stack trace to standard out. Lines 7 and 8 read and format the generated stack trace into an array of function name strings. Finally, lines 9–12 compare the execution's stack trace against the stack trace generated by the failure (stored in the BACKTRACE variable). If the stack traces are the same, the oracle returns "FAIL", indicating that the subject failure was detected, otherwise it return "UNRESOLVED", indicating that some other failure occurred. Other types of information, such as the value of program variables, can also be inspected by using this type of program introspection.

In addition to the execution's properties, the oracle must also take into consideration the execution's length of time. For applications that run for an indefinite length of time (e.g., servers or other input driven applications) it is possible that a successful execution will not terminate on its own. Oracles must be able to detect this situation. The most common technique for handling applications that wait for input is to add a time keeping mechanism to the oracle along with a cut-off value. If the execution time of the execution that the oracle is monitoring exceeds the cut-off, then the oracle will terminate the execution and indicate that no failure occurred. A more complicated approach would be to use a sampling scheme to monitor whether progress is being made by the application.

The techniques described above are sufficient for constructing oracles for all of the failures considered in this dissertation. However, as I mentioned before, this section provides a set of guidelines that I have found useful, but it is not intended to be an exhaustive resource on how to construct oracles for every failure.

3.5 Implementation

To experiment with the technique, I implemented it in a tool called ADDA (Automated Debugging of Deployed Applications).

The first implementation challenge that was faced with ADDA was how to intercept interactions between software and environment. As a first attempt at this task, the implementation focuses on interactions with the OS and does not consider interactions occurring through a windowing system. Also, I decided not to record and replay concurrency-related events, which may prevent ADDA for performing deterministic replay in some cases (but was not a problem for the studies). Because interactions with the OS occur through system-library calls, I considered two alternative approaches: operating at the system-call level or at the C-library level. For the current implementation, I decided to work at the C-library level because instrumenting at the system-call level has several drawbacks. For example, at the system-call level there is no explicit notion of parameter types and number (for function calls). Also, it is difficult to limit the amount of instrumentation code executed because it is not always possible to distinguish relevant from non-relevant calls based on the context. Instrumenting at the C-library level eliminates these and other drawbacks, but limits the applicability of the technique to software written in C or C++.

A second challenge was the mapping of C-library functions to POLL, FILE, and PULL actions. (Note that ADDA also intercepts additional function calls, such as `rand`, to account for some sources of non-determinism in complete replay. During minimized replay, these calls are not intercepted.) For many functions, the mapping is trivial. For example, functions such as `open` and `stat` map naturally to FILE actions, and functions such as `select` and `poll` map naturally to POLL actions. For other functions, however, the mapping is more complex. For example, function `read` can map to both POLL and PULL actions,

depending on the context. Luckily, the number of C-library calls to be considered for a given platform is small enough to allow for an exhaustive study of their characteristics. I studied, classified, and mapped the functions in the C library as a preliminary step towards the implementation of ADDA. ADDA uses this information to redirects to wrapping functions all relevant calls from the software to the C library, using binary instrumentation. The wrapping functions invoke the original functions while recording the information needed to update the event log and the environment data (e.g., data read from streams and timing information).

The current implementation of ADDA is based on the Pin binary instrumentation framework [54]. I chose Pin for several reasons. First, it allows for instrumenting binaries on the fly, so I can perform execution recording and replay of a software without any need to modify the software beforehand. Second, Pin can be very efficient, which is a requirement for the technique to work on real deployed software. Third, Pin works on a large number of architectures that include IA32 (Mac OS X, Windows, and Linux), IA32e (Linux), Arm (Linux), and Itanium (Linux).

The minimization component of the ADDA tool is implemented as an extensible set of Python scripts that invoke the ADDA replay tool. As discussed in Section 3.2.4, the minimization module should allow for plugging different minimization algorithms for different types of files and stream dumps. To avoid problems of bias, for the experiments the implementation used the default minimization algorithm based on delta debugging [100].

3.6 Evaluation

To assess the feasibility and effectiveness of the approach, I performed an empirical evaluation using ADDA and investigated the following research questions:

RQ1: Feasibility—Can ADDA record and replay real executions in an accurate way?

RQ2: Efficiency—How much time and space overhead does ADDA impose while recording executions?

RQ3: Effectiveness—Can ADDA automatically reduce the size of recorded failing executions and generate shorter test cases that manifest the same failures as the original executions? Can these executions still be used to debug the observed failure?

The following sections explain the experimental setup and discuss the results of the empirical evaluation.

3.6.1 Subjects

In selecting an application for the study, my goal was to create a realistic instance of the scenario shown in Figure 2. To this end, I selected `pine` [82], a widely used e-mail and news client developed at the University of Washington. I chose `pine` because it is a non-trivial program, with a large user base and, moreover, `pine`'s developers provide source code and change logs for many previous versions of the software (the earliest ones dating back to 1993).[2]

After studying several versions of `pine`, I selected version 4.63, which contains two faults that were fixed in the subsequent version. The fact that the faults were documented and fixed is a good indication that they were (1) discovered in the field and (2) considered relevant enough to be worth fixing. Because knowledge about these faults and their manifestation is important for a correct understanding of the study, I describe them in detail in the next section.

3.6.1.1 Header-color fault

This fault causes `pine` to crash if (1) color is enabled in `pine`'s configuration, (2) a user adds one or more header colors, and (3) a user removes all header colors. This fault is ideal for the study because it was not discovered during in-house testing and resulted in field failures, which is the scenario that is being targeted.

There are several ways in which a user can expose this fault and trigger the corresponding failure. Figure 6 shows three possible sequences of actions (and environment conditions) that I considered in the study. An edge between two actions A and B indicates that A must

[2] `http://www.washington.edu/pine/changes.html`

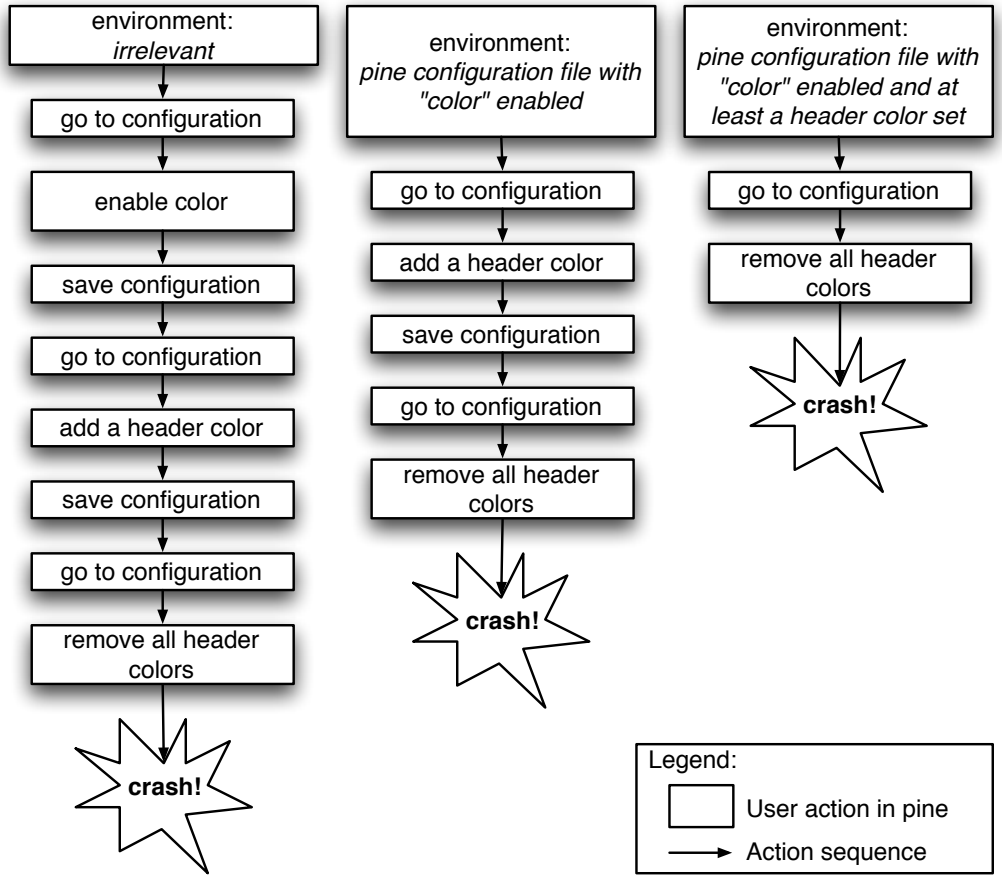

Figure 6: Possible sequences of actions that trigger the header-color fault.

occur before B, but there can be any number of different interleaving actions between the two. For example, users could read and write several email messages between the time they save the configuration file and the time they go back to the configuration to delete header colors.

3.6.1.2 Address book fault

This fault causes `pine` to crash if (1) the address book contains two entries with the same nickname and (2) the user edits the first of these entries changing it from a single address to a list of addresses. One unique characteristic of this fault, which makes it especially interesting for the evaluation, is that `pine` does not let users create two entries with the same nickname. For the fault to manifest, it is thus necessary to perform an external edit of the address book. For this fault, simply recreating user actions is not enough to reproduce

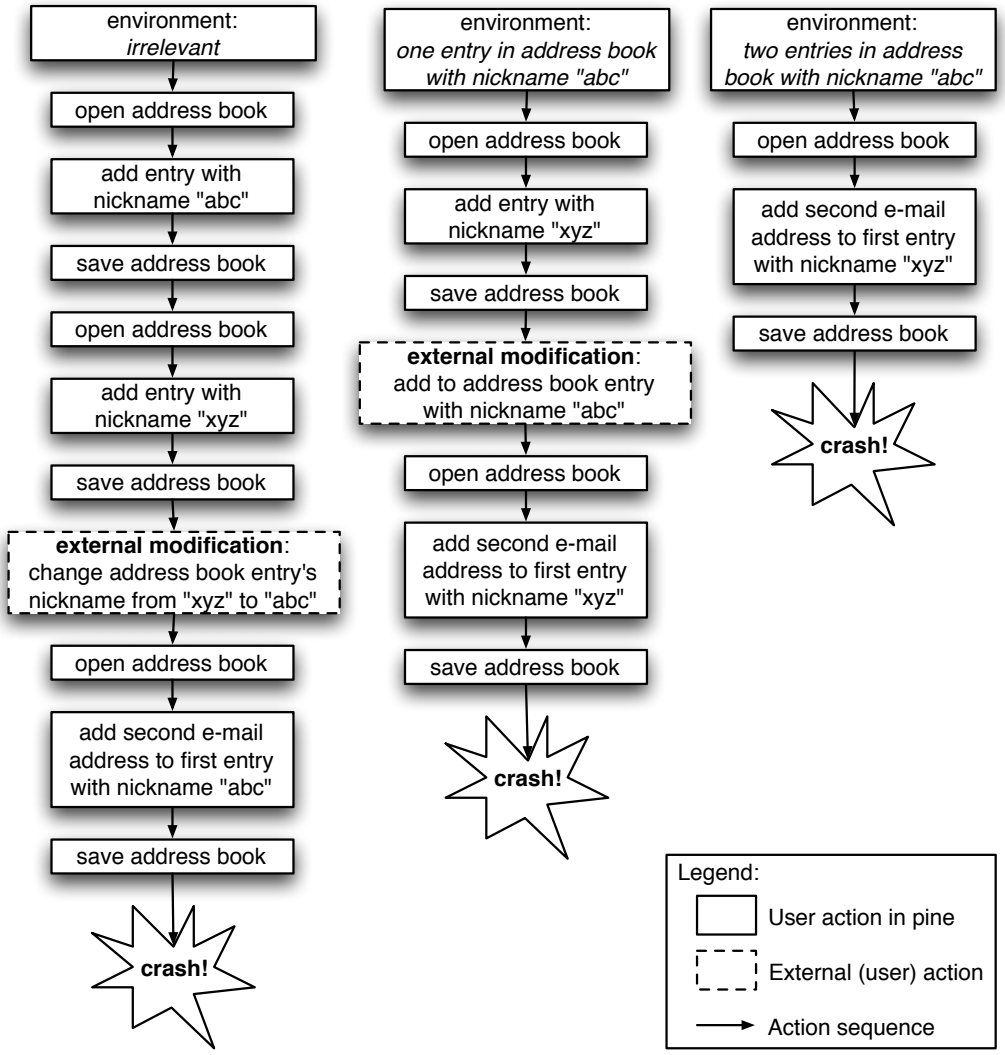

Figure 7: Possible sequences of actions that trigger the address-book fault.

the failure because the environment also plays a fundamental role in the outcome.

There are several ways in which a user can expose this fault. Figure 7 shows three possible alternatives that I considered. The figure uses the same notation as Figure 6 but also shows *external user actions*—actions performed outside of `pine` that affect `pine`'s behavior.

3.6.1.3 Collecting `pine`'s Execution Recordings

To assess replay and minimization capabilities of ADDA a set of subject executions that expose the failures considered is needed. Moreover, these executions should be rich and varied. They should contain both actions that contribute to the failure and actions that do

```
- ...
- create and send e-mail
- ...
- new e-mail received
- new e-mail received
- enable feature expanded-view-of-folders
- ...
- forward current e-mail
- ...
- remove all header colors
```

Figure 8: Excerpt of an header-color execution script used in the studies.

not contribute to the failure. Because executions captured from a sample of user sessions would be unlikely to trigger the failing behavior, and manually creating the executions could create problems of bias, I decided to generate executions in a random fashion.

Using `pine`'s manual, I compiled a list of high-level user actions. I then associated weights to actions to indicate the likelihood of users performing a given action. For example, I expect users to read and send email more often than they update `pine`'s configuration. I then wrote a simple program that produces random execution scripts of a given length. Each execution script contains, interleaved with other actions, one of the core sequences of actions shown in Figures 6 and 7.

Figure 8 shows an excerpt from an execution script. Note that the execution scripts are purposely high-level and general so that they can easily be executed by a human.

Overall, I generated 20 execution scripts (10 for each of the two faults considered), each containing between 10 and 100 high-level actions. The durations of the executions range from 5 to 30 minutes. For each execution script, I first executed the script on `pine`, run normally, and verified that the actions in the script produced the expected failure. I then executed the script again while recording the execution. At the end of this process ADDA had produced 20 execution recordings, one per script.

The next sections describe the studies I performed and their results.

3.6.2 Studies, Results, and Discussion

3.6.2.1 RQ1: Can ADDA *record and replay real executions in an accurate way?*

To address RQ1, I replayed each execution recording and checked whether the execution failed as expected. To do this, I checked that (1) the replayed execution crashed due to the same process signal as the original execution, and (2) the value of the instruction pointer was the same in the two cases. For all 20 executions, ADDA was able to correctly reproduce `pine`'s failing behavior.

To reduce the threats to external validity of this first study, I recorded and fully replayed executions of two additional subjects: BC, a numeric processing language with arbitrary precision, and GCC, a widely used C compiler. I used Version 1.06 of BC (17 KLOCs) and Version 4.0.3 of GCC (around 1,000 KLOCs).

As inputs for BC, I used several programs from the set of BC number theory programs (`http://www.numbertheory.org/gnubc/`) run with random numeric arguments. To verify whether the record and replay worked successfully, I simply checked the final results of the original and replayed executions. For GCC, I recorded the compilation of a set of student projects and used the correct termination of the compilation as the indicator of a successful replay.

For both subjects, all executions were recorded and replayed correctly to the extent that I could verify.

These results show the feasibility of the approach. In the cases considered in the study, ADDA was able to record and replay complete executions.

3.6.2.2 RQ2: How much time and space overhead does ADDA *impose while recording executions?*

I expect the technique to impose low time overhead during recording. Interactions between software and environment typically involve I/O operations that are relatively expensive and are likely to dominate the cost of the instrumentation.

It would be difficult to measure the overhead imposed by the technique on `pine`, due to its interactive nature. Although I did not experience any noticeable overhead while

recording the execution scripts, this is a qualitative and subjective assessment. To obtain a quantitative and more objective assessment of the cost of the approach, I measured the overhead imposed by ADDA on the additional two subjects used in the previous study: BC and GCC. For all executions, the difference between the execution times with and without ADDA was too small to be measured using UNIX's `time` utility. Therefore, at least for these subjects and executions, the overhead was negligible.

Despite these results, there may be cases in which the approach imposes a non-negligible overhead. In particular, if an application accesses very large files that are often changed externally, ADDA would make several copies of these files, which may involve a large time and space overhead. However, I do not expect situations of this kind to be very common. If cases where the approach is too costly are encountered, there are ways to improve its efficiency. For example, for the issue mentioned above, it may be possible to modify the technique so that it uses a differencing-based approach to save only deltas instead of complete file versions.

In the case of large input streams, the technique may start discarding part of the recorded data if the stream grows too large, using some form of minimization on the fly. I could also involve developers in the process and let them select, based on their expertise, event sources that may be simulated in-house and do not need to be recorded.

3.6.2.3 RQ3: Can ADDA *automatically reduce the size of recorded failing executions by generating shorter test cases that manifest the same failures and can be used to debug the observed failure?*

To address RQ3, I fed the 20 execution recordings for `pine` to the ADDA minimization tool, which produced 20 corresponding minimized recordings using the four steps discussed in Section 3.2.4. The minimization step took at most 75 minutes per execution. As I did for RQ1, I verified that the minimized executions exhibited the same failing behavior as the original executions by checking process signal and value of the instruction pointer at the time of the crash. Figure 9 shows a bar chart with the results for the two sets of executions.

For each set, the bar chart shows the average percentage reduction achieved by ADDA for four measurements: total size of the stream dumps (streams size), total size and number

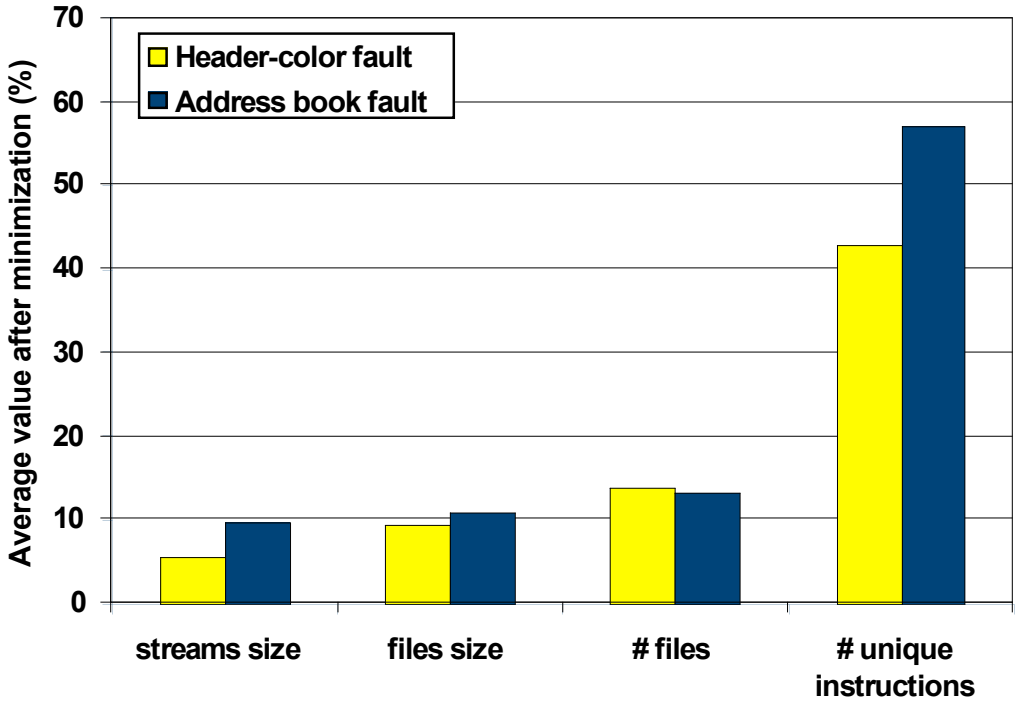

Figure 9: Results of the minimization performed by ADDA.

of environment files (files size and # files, respectively), and number of unique instructions executed during replay (# unique instructions).

As the figure shows, ADDA was able to minimize executions along all four measurements, and for three of them it achieved a considerable reduction: on average, it was able to (1) reduce the streams size by almost 95% (from about 300 to 16 characters), (2) eliminate more than 85% of the environment files (from 32 to three), and (3) reduce the total size of the environment by more than 90% (from approximately 800KB to 72KB). Whereas these first three measurements considered are indicative of ADDA's effectiveness in reducing execution recordings' size, the fourth measurement (number of unique instructions executed) is an indicator of how much the minimization can help debugging: Having fewer instructions to inspect with respect to the original failing executions has the potential to speed up debugging. As the results show, ADDA was able to generate minimized executions that exercised 42% (for the header-color fault) and 57% (for the address-book fault) of the instructions exercised by the original executions.

Note that I chose to show the reduction in the number of unique executed instructions

instead of the reduction in execution time for fairness. `pine` is driven by user actions, so its running time could increase simply due to idleness that would be eliminated during minimization and unnaturally inflate the savings. Note also that, when recording `pine`'s executions, I generated just the minimal number of inputs needed to perform the actions in the script. For example, for all e-mail-sending actions, I wrote one-line e-mails with one-word subjects. I chose this approach again to avoid an artificial inflation of the savings achieved by the technique (e.g., minimizing a 10,000-character email would certainly result in a larger reduction than eliminating a 10-character email). Because I followed the same principle wherever applicable, using ADDA on executions recorded from actual users is likely to result in even larger reductions than the ones observed in this study.

Looking at the different results, I am especially pleased by the reduction achieved by ADDA on the environment data. In almost all cases, the tool was able to completely eliminate a large percentage of files and also considerably reduce the size of the remaining files and streams. For example, there is a common group of three files that are necessary to cause the captured failures: `pine`'s configuration file, `/etc/passwd`, and `/lib/terminfo/x/xterm`. Although the technique was not able to minimize the `xterm` terminfo file, probably because of its binary content, it reduced `/etc/passwd` to a single line, the one with the entry for the email user, and `pine`'s configuration file to one or two lines, depending on the execution. Interestingly, one of these two lines is the one that records whether `pine` has been run at least once already, to avoid opening `pine`'s greeting message multiple times. When that line was removed, no execution failed because `pine` was stuck on the greetings page, waiting for an acknowledgment from the user (and none of the recorded keyboard streams contained the right key presses).

To gain some confidence that the executions recorded and minimized by ADDA can actually be used for debugging, I also performed a preliminary study on four of the 20 execution recordings. (I simply picked the first two executions generated for each fault.) For the study, I attached a well-known debugging tool, `gdb` (`http://www.gnu.org/software/gdb/`), to the replayed executions and used the tool to debug `pine` while it was being replayed by ADDA. The result of the study was successful, and I was able to perform

typical debugging actions, such as stopping at a breakpoint or watching the value of a variable, on the replayed executions.

3.6.3 Threats to Validity

Because I performed most of the studies on a single subject, the results may not generalize. However, the subject that was selected is real and widely used, the faults I considered are real faults that were not discovered during testing, and I also performed a subset of the studies on two additional subjects. Therefore, although more studies are needed, the initial results with ADDA are promising and motivate additional research in this area.

3.7 Related Work

The techniques most closely related to my approach are those that record and replay executions for testing or debugging. Some of these techniques perform deterministic replay debugging, that is, replay of executions that led to a crash (e.g., [14,44,61,62,75]). Various commercial and research tools record and replay user interactions with a software product for regression testing (e.g., [35, 76]). Unlike my approach, most of these techniques are designed to be used only during in-house testing or debugging. The overhead they impose (on time, or infrastructure required for recording) is reasonable for their intended use, but would make them impractical for use on deployed software. The few record and replay techniques that are either defined to operate in the field (e.g., BugNet [61]) or may be efficient enough to be used in the field (e.g., [44,75]) require a specialized operating-system or hardware support, which considerably limits their applicability in the short term.

Other record and replay approaches aim to reproduce, given the same inputs, the concurrent behavior of programs (e.g., [28,46,70,77]). These techniques do not need to store inputs to the application and have no efficiency constraints. Our approach is concerned with automated record and replay of user executions and has efficiency as a priority so as to be practically usable on deployed software.

More recently, several researchers presented techniques for recording executions of Java subsystems [25,63,64,71]. These approaches are heavily based on Java's characteristics and target the recording of subsystems only. It would be difficult to adapt them to work in a

more general context. Moreover, most of these approaches are defined to be used in-house, for regression testing, and impose an overhead that would prevent their use on deployed software.

Finally, none of the above record and replay techniques tries to minimize the recorded executions to improve debugging, and, because of the way in which they record executions, it would be difficult to extend them to this end.

There is also related work in the area of execution minimization. First there are techniques that attempt to minimize a single file (e.g., [93,96,100]) which are complimentary to my approach. These techniques could be used by the minimization algorithm to minimize individual files and streams. In addition to the single file techniques, Zhang and colleagues have investigated an alternate approach to minimizing executions [78, 103]. The main difference among the approaches is that they perform minimization on the captured event log, rather than the captured environment. Although their approach is able to minimize executions, it does have some drawbacks. The most significant of which is the difficulty of tracking dependencies between events in the log. Also, in some cases, their approach is limited in the amount of minimization it can perform. For example, if the event log contains a read of 50 characters, the technique's only options are to remove or keep all 50 characters. In contrast, my technique is able to minimize within the data that is read and only keep the characters that are necessary.

CHAPTER IV

AUTOMATED ANONYMIZATION OF SENSITIVE INPUTS

This chapter presents my technique for automatically anonymizing sensitive information in captured execution recordings. It is organized as follows: Section 4.1 presents a motivating example; Section 4.3 describes the technique; Section 4.4.1 describes the prototype implementation; and Section 4.4 discusses the evaluation of the technique.

4.1 Motivating Example

This section provides an example that will be used to illustrate the technique. Figure 10 shows the code for the example, which is an excerpt from a credit card processing utility that accepts Visa, American Express, and Discover credit cards. The program reads from the command line the credit card number to be processed and passes it to `isValidCardNumber`, which checks whether the provided number is valid using the Luhn formula (a simple checksumming algorithm). If the credit card number is valid, the program invokes `processCard`, which determines the type of the credit card number (i.e., Visa, American Express, or Discover) by checking the number's prefix and processes the card accordingly.

Function `processCard` contains a fault that can cause the credit card processing utility to incorrectly handle certain credit card numbers. On October 1, 2006, Discover's prefix was changed from "650" to "65." Because line 20 of `processCard` was not updated to reflect this change, valid Discover card numbers that start with "65[1–9]", such as `6521 2556 8414 3585`, are not correctly processed and cause an `UnknownCardType` exception to be thrown.

Although this program and fault are relatively simple to understand, failures caused by this fault are good examples of the type of scenario that the technique targets, for two reasons. First, such failures directly involve sensitive information (credit card numbers, in this case), which means that users would likely be unwilling to provide developers with the specific input that triggered the fault. Second, it would be difficult for commonly used approaches to provide an anonymized version of the input that still triggers the fault. In

```
   boolean isValidCardNumber(String ccn) {
 1.  if(ccn.length() != 16) return false;
 2.  int sum = 0;
 3.  boolean alternate = false;
 4.  int i = ccn.length() - 1;
 5.  for (; i >= 0; i--) {
 6.    int n = mapChar(ccn.charAt(i));
 7.    if (alternate) {
 8.      n *= 2;
 9.      if (n > 9) n = (n % 10) + 1;
10.    }
11.    sum += n;
12.    alternate = !alternate;
13.  }
14.  return (sum % 10) == 0;
   }

   void processCard(String ccn) {
15.  if(ccn.startsWith("4"))
16.    //process Visa
17.  else if(ccn.charAt(0) == '3' && (ccn.charAt(1) == '4' ||
18.                                  ccn.chatAt(1) == '7'))
19.    //process American Express
20.  else if(ccn.startsWith("650"))
21.    //process Discover
22.  else
23.    throw new UnknownCardType(ccn);
   }

   int mapChar(char c) {
24.  return (c >= '0' && c <= '9') ? c-'0' : c-'A'+10;
   }

   void main(String[] args) {
25.  if(isValidCardNumber(args[0]))
26.    processCard(args[0]);
   }
```

Figure 10: Code excerpt for the motivating example.

particular, input-minimization techniques would be likely to fail. Minimization techniques that attempt to find a subset of the inputs that causes the same failure, such as `ddmin` [100], will be unsuccessful because a valid credit card number must have 16 digits, so no minimization would be possible. Minimization techniques that perform alphabet normalization by substituting some portions of the input with a "do not care" value (e.g., `tmin` [96]) would also likely fail, as most inputs generated in this manner will not satisfy the Luhn formula. Even constructing anonymized inputs by hand would be quite challenging, due to the difficulty of generating inputs that pass the Luhn check.

4.2 Dynamic Symbolic Execution

This section briefly provides necessary background information on dynamic symbolic execution. At a high level, dynamic symbolic execution techniques (e.g., [9,10,29,73]) execute a program using symbolic inputs so that, at each point in the computation, the state is expressed as a function of the input, and the conditions on the input for reaching the current location are expressed as a conjunction of constraints called a *path condition*.

Symbolic execution techniques generate path conditions by first associating a symbolic variable v_k with every element of an input. For the example in Figure 10, for instance, each character of the command line argument would be associated with a unique symbolic variable: The first character would be associated with v_0, the second character would be associated with v_1, and so on. Then as the program executes, program statements are interpreted, and their effect determines (1) the symbolic state of the program, expressed in terms of the symbolic variables, and (2) the current path condition. As an example of a symbolic state, consider two subsequent instructions `S1`: $x = i + (j - 5)$ and `S2`: $y = x * 2$ where i is associated with a symbolic expression e, and j is associated with symbolic variable v_1; the value of y in the symbolic state after $S2$'s execution would be the symbolic expression "$(e + (v_1 - 5)) * 2$".

Path conditions are constructed incrementally, by appending a new constraint to the current path condition every time a predicate that depends on the symbolic state is executed (i.e., every time a predicate involves one or more values that have an associated symbolic

Path condition:

Constraints from mapChar

$v_0 \geq$ '0' $\wedge$ $v_0 \leq$ '9' $\wedge$

...

$v_{15} \geq$ '0' $\wedge$ $v_{15} \leq$ '9'

Constraints from processCard

$v_0 \neq$ '4' $\wedge$

$v_0 \neq$ '3' $\wedge$

$v_0 =$ '6' $\wedge$ $v_1 =$ '5' $\wedge$ $v_2 \neq$ '0'

Constraints from isValidCardNumber

((v0 - '0') * 2) > 9 ∧ ((v2 - '0') * 2) ≤ 9 ∧
((v4 - '0') * 2) ≤ 9 ∧ ((v6 - '0') * 2) > 9 ∧
((v8 - '0') * 2) > 9 ∧ ((v10 - '0') * 2) ≤ 9 ∧
((v12 - '0') * 2) ≤ 9 ∧ ((v14 - '0') * 2) > 9 ∧

((((((v0 - '0') * 2) % 10) + 1) + ((v1 - '0') + (((v2 - '0') * 2) +
((v3 - '0') + (((v4 - '0') * 2) + ((v5 - '0') + (((((v6 - '0') * 2)
% 10) + 1) + ((v7 - '0') + (((((v8 - '0') * 2) % 10) + 1) + ((v9 -
'0') + (((v10 - '0') * 2) + ((v11 - '0') + (((v12 - '0') * 2) +
((v13 - '0') + (((((v14 - '0') * 2) % 10) + 1) + (v15 -
'0'))))))))))))))) % 10) = 0

Original input: 6510 2556 8418 3585

(a) **Path condition generated by dynamic symbolic execution for the code in Figure 10 and input** 6510 2556 8414 3585**.**

Anonymized input: 6510 4546 3868 7524

(b) **Anonymized input generated by the basic approach for the path condition in Figure 11a.**

Figure 11: Path condition and anonymized input for the motivating example.

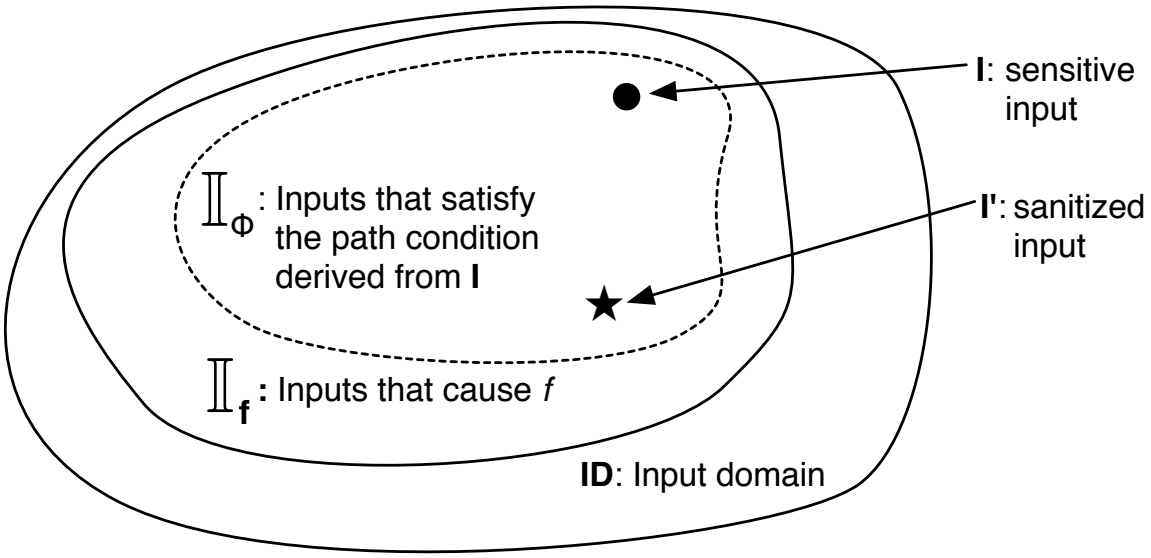

Figure 12: Intuitive view of a program domain.

expression). The constraint encodes the condition on the symbolic variables under which the predicate evaluates in the same way as the concrete execution. To illustrate, consider the code example from Figure 10, and assume that the code is run with command line argument `6521 2556 8414 3585`. When line 24 is executed for the first time, c is associated with symbolic variable v_{15}, has the concrete value $'5'$, and is compared with character $'0'$. In this case, the predicate evaluates to `true` because $'5' \geq '0'$; therefore, the constraint "$v_{15} \geq '0'$" is appended to the path condition. Had the predicate evaluated to `false`, the constraint "$v_{15} < '0'$" would have been appended instead. Figure 11a shows the complete path condition generated for this example program and input (including constraints due to library code).

4.3 Automated Anonymization

Before discussing the details of the approach, I use Figure 12 to illustrate, intuitively, the approach's goal and the context in which it operates. Given a program P with input domain ID, a failure f, and an input $I \in ID$ that results in f, there is typically a subset of the input domain, $\mathbb{I}_f \subseteq ID$, such that every input in $\mathbb{I}_f$ causes f.[1] In general, identifying $\mathbb{I}_f$ is not possible due to computability issues. However, under the assumptions that are discussed in

[1]Note that this includes the extreme (and rare) case in which $\mathbb{I}_f$ is a singleton whose only element is I.

Section 4.3.4, it is possible to identify a subset of $\mathbb{I}_f$, such that every input in this subset follows the same path as I and causes f.

The approach proposed by Castro and colleagues [11] (*basic approach*, hereafter) computes this subset using dynamic symbolic execution. Given a specific input I, dynamic symbolic execution is used to identify a subdomain of ID whose elements are inputs that cause the program to follow the same path as I. More precisely, dynamic symbolic execution is performed along the specific path of execution p caused by I; when failure f occurs, the computed path condition, ϕ, identifies this target subdomain—the set of all inputs, including I, that cause p to be executed and f to occur. This set is called $\mathbb{I}_\phi$. (Note that, in general, the fact that an input satisfies ϕ does not necessarily imply that such input will follow the same path as I or, if it does, result in f. However, this tends to be the case in most practical situations, as I discuss in detail in Section 4.3.4.) After computing ϕ, an anonymized input I' is generated by leveraging a constraint solver to identify a satisfying assignment for ϕ. For my motivating example, a possible assignment generated by this approach is shown in Figure 11b. I' can then be sent to developers who can use it to debug f. Note that because it may not be possible to remove all sensitive information, users are typically given an opportunity to inspect I' before it is sent to developers.

The strength of this approach (i.e., how well it prevents information about I from being revealed) depends on several related aspects. *First*, the solution identified by the solver, I', should be independent from I (i.e., it should not be possible to algorithmically recover I from I'). *Second*, $\mathbb{I}_\phi$ must be large enough to make an enumeration of the domain impractical in a reasonable amount of time. (Because ϕ can be derived from I', just as it is derived from I, a sufficiently small domain allows for easily recovering I, which defeats the purpose of the technique.) And *third*, the inputs in I' should be as different as possible from their corresponding inputs in I. Although, as mentioned previously, there are cases where only a single value can satisfy some constraints in ϕ (e.g., in my example, v_0 must be equal to $'6'$), in general, trying to prevent I' from duplicating I reduces the likelihood that sensitive information can be identified simply by examining I'.

I expect the first aspect not to be a problem; most constraint solvers utilize some

randomness in their search heuristics, so the selection of I' can be safely considered pseudo-random. To address the second and third aspects, and thus improve the strength of the approach, I extend the basic approach by means of two novel techniques: path condition relaxation and breakable input conditions. *Path condition relaxation* addresses the second aspect—the size of $\mathbb{I}_\phi$. It consists of a set of optimizations that specialize the constraint generation part of dynamic symbolic execution to increase the size of $\mathbb{I}_\phi$. Intuitively, the technique relaxes overly restrictive constraints, thus strengthening the overall approach by allowing for a larger number of solutions. *Breakable input conditions* address the third aspect by forcing the constraint solver to chose, whenever possible, values for I' that are different from the corresponding values in I.

The rest of this section presents pseudocode for the anonymization algorithm and discusses in detail: path condition relaxation, breakable input conditions and the assumptions that underlie the overall approach.

4.3.1 The Anonymization Algorithm

Algorithm 2 Pseudocode describing how failure-inducing inputs are anonymized.

Input: P: Subject program.
Input: I: Sensitive input that causes failure f to manifest in P
Output: An anonymized input that causes failure f to manifest in P or $\varnothing$ if an anonymized input could not be generated.

```
1:  procedure AnonymizeInput(P, I)
2:      M = ∅                                  ▷ Map from inputs to symbolic variables
3:      BIC = ∅                                ▷ Set of breakable input constraints
4:      for i ∈ I do
5:          M = M + [i ↤ makeSymbolic(i)]
6:          BIC = BIC ∪ ⟨M(i) ≠ valueOf(i)⟩
7:      end for

8:      PC = executeSymbolic(P, I, M)          ▷ Generate the path condition for the failure

9:      I' = solve(PC, BIC)                    ▷ Solve the generated constraints
10:     return I'
11: end procedure
```

Algorithm 2 presents pseudocode describing the input anonymization algorithm. As input the algorithm takes a subject program P and a set of sensitive inputs I that causes

a specific failure f in P. Lines 4–7 are responsible for initializing two data structures: $\mathcal{M}$ and $\mathcal{BIC}$. $\mathcal{M}$ is a mapping from each input in I to its corresponding symbolic variable and $\mathcal{BIC}$ is the set of breakable input conditions that attempt to prevent the symbolic variable from having the same value as the input.

After the initialization steps, the path condition for failure f is generated by calling *executeSymbolic* at line 8. Program P is symbolically executed with input I and mapping $\mathcal{M}$. Except for the modifications described in Section 4.3.2, the symbolic execution used by the algorithm is identical to the approach used by many other symbolic execution based techniques (e.g., [9, 29, 73]).

Finally, after generating the path condition, an anonymized input I' is generated by solving the conjunction of the path condition and breakable input constraints (line 9). The *solve* function is responsible for interacting with the underlying constraint solver. Its main responsibilities are to translate $\mathcal{PC}$ and $\mathcal{BIC}$ into a format that the solver can parse and then to translate the solver's solution, if any, back into an executable input. If no solution can be found, solve returns $\varnothing$. Note that because, by definition, the path condition and breakable input constraints have a solution (i.e., the original input values) it is unlikely that the solver will be unable to satisfy the constraints. Details about the solve function for my prototype implementation are provided in Section 4.4.1.

4.3.1.1 Characteristics of the Anonymization Algorithm

Worst-case execution time. The most significant contributor to the runtime of the algorithm is the *solve* function. The initialization done in lines 4–7 and the generation of $\mathcal{PC}$ are linear with respect to the subject program and the original input; one symbolic variable is constructed for each input and the program is executed a single time to generate the path condition.

In essence, the *solve* function is a wrapper around the underlying constraint solver. (See Section 4.4.1 for a description of the steps the algorithm performs before invoking the solver) As such, the algorithm's worst-case runtime and memory performance are dependent on the solver. In general, SMT solvers have a worst-case runtime complexity that is exponential

in the size of the input formula (i.e., the number of variables and constraints in $\mathcal{PC}$ and $\mathcal{BIC}$) [23]. However, this worst case is rarely encountered in practice [23]; an observation that is also demonstrated by my evaluation in Section 4.4.3.

Soundness and Completeness. The anonymization algorithm is sound in the sense that if it generates an anonymized input, that input will reproduce the failure (given the assumptions in Section 4.3.4). Like for execution time, this property is based primarily on the use of a constraint solver that is sound (i.e., if a a solution is found, it is guaranteed to satisfy the constraints). Similarly, the algorithm is not complete. It may be possible that the solver can fail to find a satisfying assignment even if one exists. For example, this may happen if the constraints include complex exponential expressions.

Note that the algorithm makes no guarantees about the quality or effectiveness of the anonymization; the algorithm only makes a best effort to remove sensitive information. However, as I show in Section 4.4, the algorithm can be effective at anonymizing failure-inducing inputs.

4.3.2 Path Condition Relaxation

As I mentioned in the previous section, path condition relaxation consists of several optimizations that modify the way path conditions are generated, so as to increase the number of solutions for the computed conditions. In the following, I describe four cases that the technique optimizes. Note that, because these optimization are designed for my specific goal of finding inputs that cause a known path to be executed, I believe that they are unlikely to benefit other dynamic symbolic execution techniques (e.g., [9, 29, 73]) in any significant way.

Array inequality: Typically, a comparison between two arrays is performed by iterating over the arrays and performing a pairwise comparison between corresponding elements. In traditional symbolic execution, the result of each comparison would be recorded as a constraint in the path condition. Therefore, only inputs that cause every comparison to evaluate the same way as the observed execution would satisfy the

path condition. For example, assume that $a = [1, 2, 3]$, $b = [1, 2, 4]$, a's elements are associated with symbolic expressions e_1, e_2, and e_3, and b's elements are associated with symbolic expressions e_4, e_5, and e_6. Checking the equality of these arrays would add the constraints "$e_1 = e_4$", "$e_2 = e_5$" and "$e_3 \neq e_6$" to the path condition.

The key intuition behind the optimization of array comparisons is that such comparisons are essentially atomic operations. Therefore, when arrays are not equal, path condition relaxation can replace the constraints that encode the individual comparisons with a constraint that simply requires that at least one comparison evaluates to `false` (i.e., at least one element is different). For the previous example, the constraint "$e_1 \neq e_4 \vee e_2 \neq e_5 \vee e_3 \neq e_6$" would be added. All variable assignments that satisfy the original constraints also satisfy the relaxed one, but the latter is also satisfied by many other assignments (e.g., $a = [2, 2, 3]$, $b = [1, 2, 4]$).

Multi-clause conditionals: In many languages, commonly used Boolean operators such as "and" and "or" are evaluated with short-circuit or minimal evaluation semantics; only the minimal amount of evaluation is done to determine the value of the expression. For example, consider the conditional "`if`$(a_1 > 5 \parallel a_2 > 5)$", where a_1 and a_2 are associated with symbolic expressions e_1 and e_2, respectively. Using a left-to-right evaluation order, if a_1's value is 6, then "$a_2 > 5$" will not be evaluated, as the outcome of the condition is known after the evaluation of "$a_1 > 5$". Because the rest of the conditional is not evaluated, the path condition will only include the constraint "$e_1 > 5$". Like for array inequalities, such path conditions exclude a large number of assignments that would cause the conditional to evaluate to the same value (e.g., $a = 0$, $b = 6$).

To generate relaxed path conditions for multi-clause conditionals, the technique generates constraints that encode all clauses in the conditional, not just those evaluated at runtime. For example, if the aforementioned conditional "`if`$(a_1 > 5 \parallel a_2 > 5)$" were to evaluate to `true`, the constraint "$e_1 > 5 \vee e_2 > 5$" would be generated. Conjunctive clauses and conditionals comprised of more than two clauses are handled

in a similar manner.

Switch statements: Switch statements are similar to multi-clause conditionals in that multiple values can cause the switch to jump to the same target (i.e., for case statements that immediately fall through to their successor). In these cases, a traditional technique would build a constraint of the form "$e_i \neq c_1 \wedge \cdots \wedge e_i \neq c_{m-1} \wedge e_i = c_m$", where e_i is the symbolic expression associated with the value compared by the switch statement, c_m is the value in the first case statement that matches e_i, and $c_1, \ldots, c_{m-1}$ are the values of the cases that precede c_m in the switch statement. Conversely, the technique would generate the constraint "$e_i \neq c_1 \wedge \cdots \wedge e_i \neq c_{m-1} \wedge (e_i = c_m \vee \cdots \vee e_i = c_n)$", where $c_m, \ldots, c_n$ are the values of the cases that branch to the same target as c_m. In this way, the technique allows more possible values of e_i to be considered.

Array reads: When symbolically executing array accesses, concretization[2] is often performed. For example, assume that $a = [2, 0, 1]$ and that x has the value "0" and is associated with symbolic expression e when statement "`if`$(a[x] > 0)$" is executed. In the common case of a solver that cannot handle symbolic array indices, e would be concretized to "0". In these cases, concretizing may be unnecessarily restrictive, as multiple values in an array could satisfy the same condition. In my example, the values "2" at index 0 and "1" at index 2 would both cause the conditional to evaluate to `true`, which means that e can be equal to either "0" or "2". To generate these extended path conditions, the technique uses an approach similar in nature to the one proposed by Elkarablieh and colleagues [27]. Essentially, a snapshot of the contents of the array is encoded in the path condition as a sequence of ternary-expressions. Considering again my example, the technique would generate the constraints "$((e == 0)\ ?\ 2 : (e == 1)\ ?\ 0 : 1) > 0\ \wedge\ e \geq 0\ \wedge\ e < 3$". These constraints ensure that the value of e is within the bounds of a, yet allow it to be any value that satisfies the conditional.

[2] *Concretization* [29,73] is a technique used in dynamic symbolic execution to handle constraints that the decision procedure in the constraint solver does not support. It works by replacing symbolic expressions in the problematic constraints with their corresponding concrete values, so that the constraints become decidable.

Path condition:

Constraints from mapChar and isValidCardNumber
same as Figure 2

Constraints from processCard (relaxed)

$v_0 \neq$ '4' $\wedge$
($v_0 \neq$ '3' $\vee$ ($v_1 \neq$ '4' $\wedge$ $v_1 \neq$ '7')) $\wedge$
$v_0 =$ '6' $\wedge$ $v_1 =$ '5' $\wedge$ $v_2 \neq$ '0')

Breakable input conditions

$v_0 \neq$ '6' $\wedge$ $v_1 \neq$ '5' $\wedge$ $v_2 \neq$ '1' $\wedge$ $v_3 \neq$ '0' $\wedge$ $v_4 \neq$ '2' $\wedge$
$v_5 \neq$ '5' $\wedge$ $v_6 \neq$ '5' $\wedge$ $v_7 \neq$ '6' $\wedge$ $v_8 \neq$ '8' $\wedge$ $v_9 \neq$ '4' $\wedge$
$v_{10} \neq$ '1' $\wedge$ $v_{11} \neq$ '8' $\wedge$ $v_{12} \neq$ '3' $\wedge$ $v_{13} \neq$ '5' $\wedge$ $v_{14} \neq$ '8' $\wedge$
$v_{15} \neq$ '5'

Original input: 6510 2556 8418 3585
Anonymized input: 6538 7578 2506 9852

Figure 13: Path condition and anonymized input generated by my approach for the example in Figure 10 and input 6510 2556 8414 3585.

To illustrate, Figure 13 shows how the above optimizations would relax the constraints generated for the motivating example. Although in this case the narrow constraints on the first two digits of the input nullify the effects of relaxation, in general such relaxation can result in the expansion of the set of possible solutions, as shown empirically in Section 4.4.6.

4.3.2.1 The safety of path condition relaxation

Path condition relaxation is safe in the sense that the constraints it generates describe the same path as the constraints generated by traditional symbolic execution. By definition, solutions generated from relaxed path conditions will induce the same branch outcomes as solutions generated from basic symbolic execution. In essence, path condition relaxation allows the solver to choose from a wider range of values that induce the same path, but it does not change the path that is executed.

4.3.3 Breakable Input Conditions

The use of a constraint solver to compute I' by solving the path condition for a failing execution guarantees that I' can reproduce the failure considered. However, because the solver's only goal is to find a satisfying assignment, it may coincidentally choose values that are in I, even when other choices are possible, thus unnecessarily revealing information about the original input. This is especially true when the domain of the input is small. In the example, for instance, there are only ten possible values for each digit, and the domain is further reduced by the constraints on the inputs. It is not surprising that, in a similar situation, an anonymized credit card number could share many digits with the original number. The anonymized input shown in Figure 11b, where the unchanged values are underlined, illustrates an instance of this problem, which I have also observed in practice in the empirical evaluation (see Section 4.4.6).

To address this issue, I add to the path condition *breakable input conditions*—constraints that explicitly prevent the constraint solver from choosing values from I. More precisely, for each symbolic variable, the technique adds a new constraint that specifies that the symbolic variable should not be equal to the original value of its corresponding input. For the running example, the breakable input conditions for the given input are shown in Figure 13.

When the conjunction of a path condition and the corresponding breakable input conditions is satisfiable, and the solver is able to find a solution for the resulting set of constraints, I' is guaranteed to be different from I. In situations where an element must have a specific value in order to cause the failure, however, the breakable input conditions make the path condition unsatisfiable. This is the case for the example, where the first two digits of the input must be "65", and the other digits must pass the Luhn checksumming algorithm. For this reason, the technique encodes each breakable input condition as a discardable constraint, that is, a constraint that the solver can ignore in order to find a solution for an otherwise unsatisfiable set of constraints. Because the solver would still try to ignore as few discardable constraints as possible, encoding breakable input conditions as discardable constraints forces the solver to choose I' such that it is as different as possible from I.

Figure 13 shows an example of an anonymized input computed using breakable input conditions, where only three values are shared with the original input—the minimum number of values necessary to recreate the failure and pass the checksumming algorithm.

4.3.4 Assumptions

The technique is based on two main assumptions about failure f: (1) f is observable and can be encoded in the form of an assertion, and (2) any input that satisfies path condition ϕ not only follows the same path as the original failure-inducing input I, but also results in the same failure f.

The first assumption is common to all debugging-related techniques and holds in most, if not all, cases. The second assumption requires that the necessary conditions for f are encoded in ϕ. The only cases in which this assumption does not hold are non-determinism and implicit checks. If the program being considered is non-deterministic, the technique may not be able to generate anonymized inputs that reproduce the failure because it cannot guarantee that events such as thread switches always occur in the same order. This problem is also common to all debugging-related techniques and can be addressed by leveraging record/replay infrastructure that supports deterministic replay (e.g., [87]).

Implicit checks are checks that are performed by an entity that is external to the application and, thus, are not observable by the symbolic execution. A typical example of implicit checks is the set of checks performed by the underlying runtime system, such as checks that may result in a division-by-zero or out-of-memory error.

Because these checks are not performed by the application, they would not be included as constraints in the path condition ϕ, and an input that satisfies ϕ may fail to reproduce f. Although this issue exists, I believe it is of limited relevance in most cases, for two reasons. First, although (some types of) implicit checks occur frequently, the majority of them are likely to be irrelevant because, as confirmed by the evaluation, they constraint variables that are not directly or indirectly related to the failure. Therefore, in the worst case, the technique could simply ignore cases for which the anonymized input cannot reproduce f and focus on the remaining failures. Second, it is possible to automatically account for

implicit checks that occur within the runtime system by making them explicit. To account for division-by-zero errors, for instance, a developer could easily use an automated tool to preprocess the code and add an explicit check of the denominator's value every time a division is encountered.

4.4 Evaluation

To evaluate the technique, I implemented it in a prototype tool, called CAMOUFLAGE, and investigated the following research questions:

RQ1: Feasibility—Can the approach generate, in a reasonable amount of time, anonymized inputs that reproduce the original failure?

RQ2: Strength—How much information about the original failure-inducing inputs is revealed by the approach?

RQ3: Effectiveness—Are the anonymized inputs generated by my approach safe to send to developers?

RQ4: Improvement—Does the use of path condition relaxation and breakable input conditions provide any benefits over the basic approach?

Note that RQ2 provides an objective assessment of the technique; it does not make any assumptions about whether the revealed information is actually sensitive. Conversely, RQ3 does take into account whether the information that is revealed is indeed sensitive.

The remainder of this section discusses CAMOUFLAGE, the subjects, and my experimental protocol and results.

4.4.1 Prototype Tool

Our CAMOUFLAGE tool is a prototype implementation of the technique for applications written in the Java language. It consists of two separate components: the constraint generator, which implements lines 1–8 of Algorithm 2, and the input anonymizer which implements line 9 of Algorithm 2,. (We consider the record/replay tool that would provide inputs to be anonymized to CAMOUFLAGE as an external component.) I implemented the current

version of the constraint generator as an extension to Java PathFinder (JPF), an explicit state software model checker for Java software[3]. The details of the implementation are described below.

To assign symbolic variables to an application's inputs, I use JPF's method interception capabilities to wrap all native methods in the `java.io` package. Because, ultimately, all file and network inputs are read by these methods, CAMOUFLAGE can easily associate a symbolic variable with every input read from these sources. To handle other sources of input, I also wrap the `main` method (to handle command line arguments) and the appropriate methods for reading environment variables and system properties. By default, CAMOUFLAGE assumes that all inputs are sensitive. However, it also allows users to specify that inputs read from specific sources should not be associated with a symbolic variable. This feature is useful, for example, in cases where it is known that inputs read from certain files or network streams are not sensitive and do not need to be anonymized. To implement the specialized path condition generation (see Section 4.3.2), I use JPF's bytecode overloading facilities to replace each Java bytecode with a modified version that replicates the instruction's original semantics while also performing the necessary steps for generating path conditions. Finally, to identify when failures occur, I use JPF's VMListener interface to intercept uncaught exceptions and failed assertions. When the execution reaches the point of failure, and the failure occurs, the constraint generator writes the recorded path condition and the breakable input conditions to disk.

The input anonymizer is implemented as a set of Ruby scripts and works as follows. First, it transforms the constraints produced by the constraint generator into a format understood by the constraint solver. Then, it invokes the constraint solver to find a solution for the constraints. Finally, it transforms the solution provided by the constraint solver into a concrete input that can be sent to developers. As the constraint solver, I choose YICES [24] because, among the constraint solvers that I know, it is the only one that supports both discardable constraints (see Section 4.3.3) and bit vector operations. Using bit vectors for symbolic variables allows the implementation to handle bit shifts and masks, which are

[3] `http://javapathfinder.sourceforge.net/`

commonly used in the Java libraries. However, using bit vectors does have one drawback: currently, no constraint solver, including YICES, supports floating point arithmetic on bit vectors. This means that, currently, CAMOUFLAGE does not support symbolic floats or doubles.

4.4.2 Subjects

The goal of the technique is to generate anonymized inputs that cause the same failures as the original input while revealing as little information as possible. To suitably evaluate the technique with respect to this goal, I selected applications with known faults that process information that can be considered private or sensitive: `NanoXML` (16 faults), which is available from SIR (Software-artifact Infrastructure Repository),[4]; a Java version of `printtokens` (2 faults), whose original C implementation is also available from SIR; the address book component of the `Columba` email client version 1.4[5] (1 fault); and version 1.0 of `htmlparser`[6] (1 fault). For each fault, I selected multiple failure-inducing inputs. For `NanoXML` and `printtokens`, I used the inputs provided with the two applications. For `Columba` and `htmlparser`, I constructed representative inputs by hand. In total, I used 170 failure-inducing inputs that range in size from several hundred bytes to over five megabytes.

4.4.3 RQ1: Feasibility

The goal which motivates the first research question is to assess whether the amount of time needed to generate anonymized inputs is reasonable and whether the anonymized inputs reproduce the original failure. To generate the data necessary for investigating these questions, I proceeded as follows: For each failure-inducing input, I used CAMOUFLAGE to run the application and generate an anonymized version of such input. In addition, I recorded two measurements: (1) the amount of time needed by CAMOUFLAGE to generate the path condition and (2) the amount of time needed by the constraint solver to solve the generated path condition.

The top-half of Figure 14 presents a bar chart that shows, for each fault, the average

[4] `http://sir.unl.edu/`
[5] `http://www.columbamail.org`
[6] `http://htmlparser.sourceforge.net`

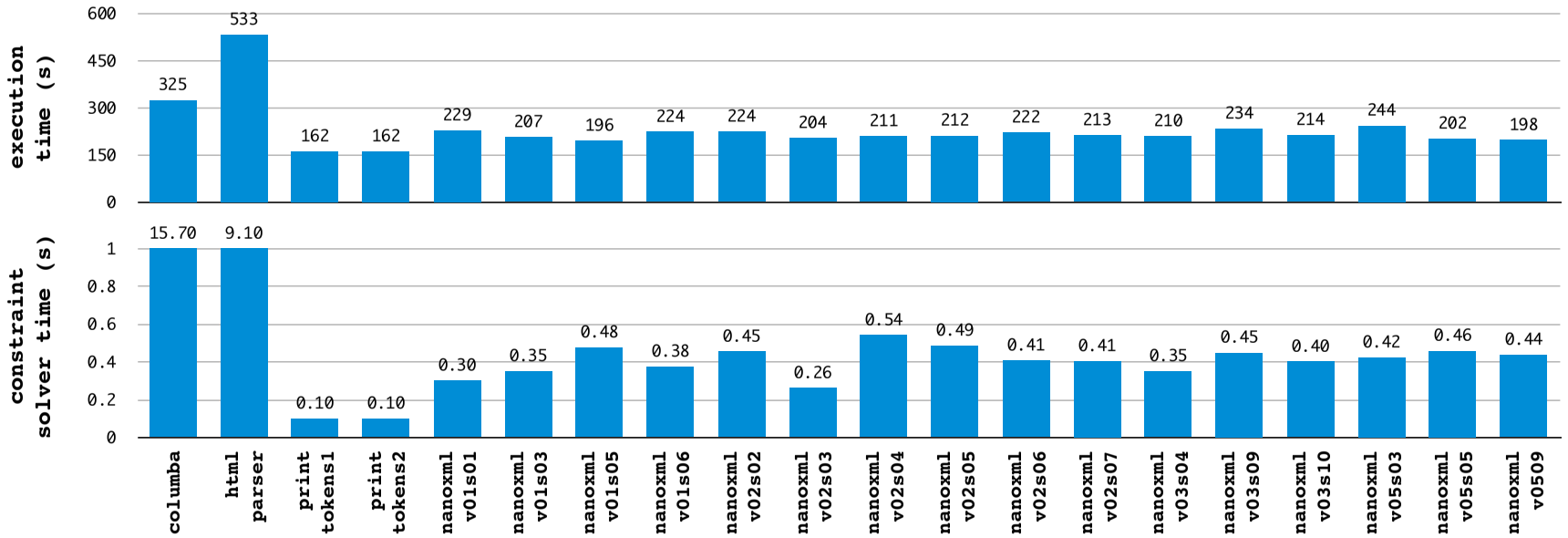

Figure 14: Bar charts showing, for each fault, the average amount of time needed to execute the subject and generate the corresponding path condition (top) and average amount of time needed for the constraint solver to find a solution (bottom).

amount of time CAMOUFLAGE needed to generate path conditions. The bottom-half of the figure shows the average amount of time needed by the constraint solver to solve the generated path conditions. As the figure shows, the amount of time needed to generate path conditions ranges from an average of 162 seconds (for `printtokens`) to an average of 533 seconds (for `htmlparser`). The amount of time needed to solve the path conditions ranges from an average of 0.1 seconds (for `printtokens`) to an average of 15.7 seconds (for `Columba`). Overall, for all of the failure-inducing inputs that I considered, CAMOUFLAGE was able to generate an anonymized version in less than 10 minutes. Because CAMOUFLAGE is designed to run off-line, during idle periods when free cycles are available (e.g., overnight), the approach is clearly practical. Users will only experience the overhead caused by the record/replay technique used, which have been shown to be in the single digits for modern approaches [18, 87].

To determine whether the anonymized inputs reproduce the original failures, I executed the subject applications with such inputs and manually inspected the outcomes. I found that all 170 anonymized inputs produced by CAMOUFLAGE successfully reproduced the original failure.

4.4.4 RQ2: Strength

To assess the strength of CAMOUFLAGE's anonymization, I used two metrics: bits of information revealed and residue. The first metric, *bits of information revealed*, is a standard entropy measure that has been used in related work [11, 88]. Intuitively, it measures how much information is revealed by the technique by calculating how many inputs satisfy the path condition (i.e., the number of inputs in $\mathbb{I}_\phi$). In general, an anonymized input reveals $\sum_{i \in I'} |\log_2(x_i)|$ bits of information about I, where x_i is the number of solutions to the constraints involving i divided by the size of i's input domain. For example, assume that i'_0 is an 8-bit character (i.e., its input domain contains 256 values) and that 5 of the 256 possible values satisfy the constraints on i'_0. In this case, i'_0 reveals approximately 5.76 of the 8 total bits of information about i_0. Because computing x_i exactly is difficult and expensive when constraints involve multiple input elements, I chose to use an algorithm by

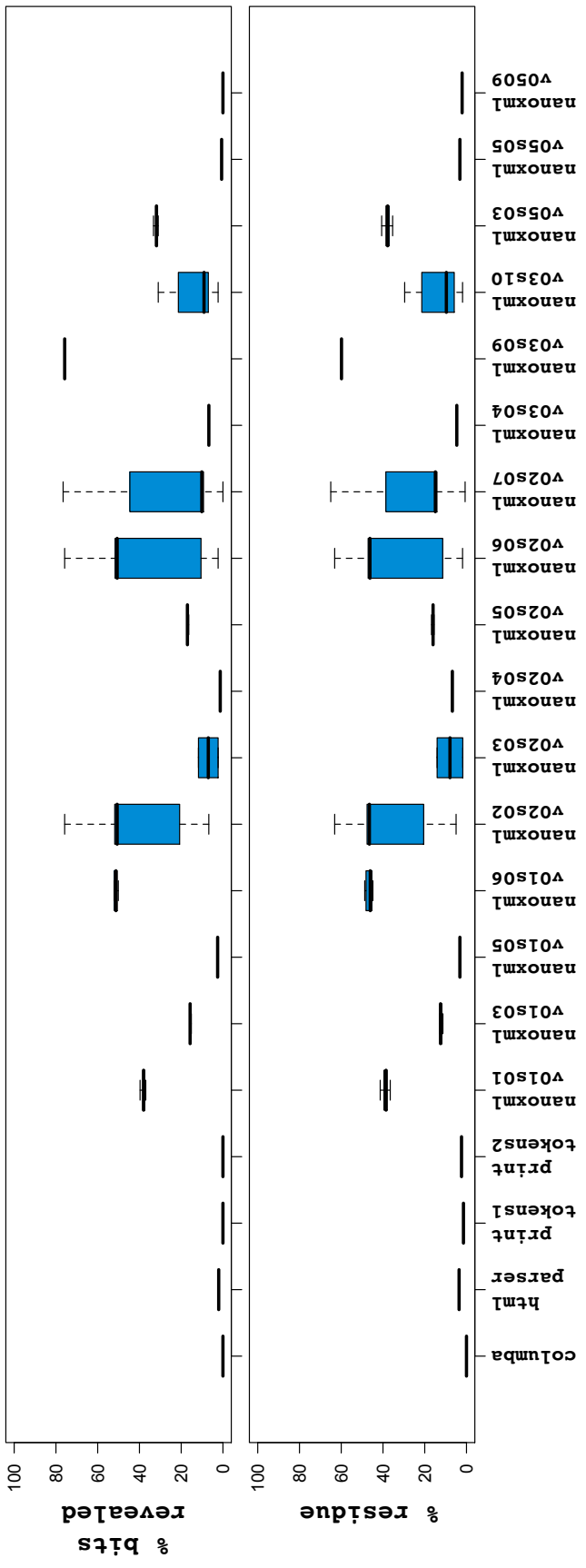

Figure 15: Box plots showing, for each fault, the bits of information revealed as a percentage of the total number of bits in the input (top) and the percentage of residue (bottom) that remains after anonymization.

Martin that quickly provides an accurate over-approximation for x_i [55].

The bits-of-information-revealed metric provides a good starting point for assessing the strength of the anonymization. However, its results can be misleading. For example, it is possible to decrease the amount of bits revealed while large portions of the input remain unchanged. To illustrate this situation, consider a program that reads 10 characters as input. Assume that the constraints on each of the last 5 characters have 10 possible solutions, while the first 5 characters must remain the same. If the number of possible solutions for the second 5 characters is increased from 10 to 200, the amount of information revealed decreases from 63.3 bits to 41.7 bits. This decrease correctly indicates that it is now more difficult to recover the original input, but it fails to indicate that half of the input is unchanged, a fact that may be important, especially if the first half of the input is more sensitive than the second half.

The second metric, *residue*, addresses this shortcoming. Residue is essentially the number of inputs that remain unchanged after anonymization. For the example mentioned in the previous paragraph, the percentage of residue would not change if the number of possible solutions for the second 5 characters increased from 10 to 200, thus indicating that anonymization may not have been as effective as the bits of information revealed metric would suggest. By using both metrics, I can assess the strength of the anonymization performed by CAMOUFLAGE from multiple perspectives and better judge how much information about the failure-inducing inputs is revealed by their anonymized versions.

Figure 15 presents two box-and-whisker plots that show, for each fault and failure-inducing input, the bits of information revealed by the anonymized input as a percentage of the total number of bits in the failure-inducing input (top) and the percentage of residue in the anonymized input (bottom). For the subjects I considered, the average percentage of bits of information revealed ranges from 2.3% to 76.5%, with an average of 30.6%, and the average percentage of residue ranges from 1.5% to 65%, with an average of 30%.

Although these results confirm that it is generally impossible to generate fully anonymized inputs, they are also encouraging; the majority of anonymized inputs produced by CAMOUFLAGE reveal only a limited amount of information. (Moreover, as the results discussed

in the next section show, the information revealed is unlikely to be sensitive.) These results also suggest that the strength of the anonymization performed by CAMOUFLAGE depends not only on the subject application, but also on the specific input considered and can vary widely even among different inputs that trigger the same fault.

4.4.5 RQ3: Effectiveness

The results of RQ2's investigation provide an objective measure of the sanitation performed by CAMOUFLAGE. However, without considering whether the revealed information is actually sensitive, it is difficult to accurately assess if anonymized inputs can safely be sent to developers. Performing such an assessment is the goal of the study addressing RQ3. In this study, I conducted an in-depth, qualitative assessment of all the anonymized inputs generated by CAMOUFLAGE that takes into account whether the revealed information is sensitive. To make this determination, I manually inspected each failure-inducing input and its anonymized version. (As the discussion of the specific anonymization cases in the rest of this section will show, for the subjects I considered the distinction between sensitive and not sensitive inputs was fairly clear-cut.) For all 170 anonymized inputs, I found that they did not reveal any information that I believe to be sensitive. In the rest of this section, I provide a detailed description of the analysis for three anonymized inputs: one for `NanoXML`, one for the address book component of `Columba`, and one for `htmlparser`. I chose to present these inputs because, among the failure-inducing inputs for each application, they have the highest percentage of bits of information revealed and residue. Consequently, they are the most likely to actually reveal sensitive information.

Figures 16, 17, and 18 show representations of the portions of the original inputs that can and cannot be changed (i.e., residue) for the inputs I am presenting. (I slightly shortened the inputs to make them fit.) In these figures, portions of the inputs that can be changed are colored gray, while portions that cannot be changed are colored black.

NanoXML. The input for `NanoXML` shown in Figure 16 is an XML file available from the SIR repository. The fault triggered by this input causes `NanoXML` to incorrectly handle closing tags. As the figure shows, the portions of this file that cannot be changed do not

```
<!DOCTYPE Foo [
   <!ELEMENT Foo (ns:Bar)>
   <!ATTLIST Foo
       xmlns CDATA #FIXED 'http://nanoxml.n3.net/bar'
       a     CDATA #REQUIRED>

   <!ELEMENT ns:Bar (Blah)>
   <!ATTLIST ns:Bar
       xmlns:ns CDATA #FIXED 'http://nanoxml.n3.net/bar'>

   <!ELEMENT Blah EMPTY>
   <!ATTLIST Blah
       x    CDATA #REQUIRED
       ns:x CDATA #REQUIRED>
]>
<!-- comment -->
<Foo a='test' b='test1' c='test2'>vaz
   <ns:Bar>
       <Blah x="1" ns:x="2"/>
   </ns:Bar>
</Foo>
```

Figure 16: Failure-inducing input for `NanoXML`

contain any sensitive information. The literals "DOCTYPE", "ATTLIST", and "FIXED" are keywords of the language used to specify document type definitions, and `NanoXML` specifically checks for their presence. Similarly, the angle brackets, exclamation points, hyphens, double quotation marks, backslashes, and equals signs that cannot be changed are necessary because they define the structure of the XML document. Conversely, portions of the input that are likely to contain sensitive information, such as XML tag names, attribute values, and tag bodies, can all be changed without preventing the modified input from reproducing the failure. Therefore, although a relatively large percentage of the file cannot be changed ($\approx 65\%$), I can consider the input to be anonymized because it contains, to the best of my knowledge, no real sensitive information.

Columba. The input for `Columba` shown in Figure 17 is a comma-separated-value file of contact information. The entries in each row are a contact's first name, last name, sort

```
...
Wayne,Bartley,Bartley,Wayne,wbartly@acp.com,,
Ronald,Kahle,Kahle,Ron,ron.kahle@kahle.com,,
Wilma,Lavelle,Lavelle,Wilma,,lavelle678@aol.com,
Jesse,Hammonds,Hammonds,Jesse,,hamj34@comcast.com,
Amy,Uhl,Uhl,Amy,uhla@corp1,com,uhla@gmail.com,
Hazel,Miracle,Miracle,Hazel,hazel.miracle@corp2.com,,
Roxanne,Nealy,Nealy,Roxie,,roxie.nearly@gmail.com,
Heather,Kane,Kane,Heather,kaneh@corp2.com,,
Rosa,Stovall,Stovall,Rosa,,sstoval@aol.com,
Peter,Hyden,Hyden,Pete,,peteh1989@velocity.net,
Jeffrey,Wesson,Wesson,Jeff,jwesson@corp4.com,,
Virginia,Mendoza,Mendoza,Ginny,gmendoza@corp4.com,,
Richard,Robledo,Robledo,Ralph,ralphrobledo@corp1.com,,
Edward,Blanding,Blanding,Ed,,eblanding@gmail.com,
Sean,Pulliam,Pulliam,Sean,spulliam@corp2.com,,
Steven,Kocher,Kocher,Steve,kocher@kocher.com,,
Tony,Whitlock,Whitlock,Tony,,tw14567@aol.com,
Frank,Earl,Earl,Frankie,,,
Shelly,Riojas,Riojas,Shelly,srojas@corp6.com,,
...
```

Figure 17: Failure-inducing input for `Columba`.

key, nickname, email address, work phone, and home phone (albeit none of the entries shown actually has a phone number). This file triggers a fault in a section of `Columba` that handles the email portion of each row. `Columba` assumes that each contact has an email address. If this assumption is violated, as it is by the second-to-last row in the part of the input shown in Figure 17, an exception is thrown. The results of anonymizing this input are similar to the results of anonymizing the input for `NanoXML`; the structural elements of the file (i.e., the commas that separate the individual fields) cannot be changed, but the non-structural elements (i.e., each contact's first name, last name, and so on) can all be changed. Consequently, also in this case, I can conclude that the input produced by CAMOUFLAGE is anonymized (i.e., contains no sensitive information) and can be safely sent to developers.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://
www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>james clause @ gatech | home</title>

<style type="text/css" media="screen" title="">
<!--/*--><![CDATA[<!--*/

  body {
    margin: 0px;
...

/*]]>*/-->
</style>
</head>
<body>
  ...
</body>
```

Figure 18: Failure-inducing input for `htmlparser`.

HtmlParser. The input for `htmlparser` shown in Figure 18 is an HTML file taken from the author's website. The fault that this input triggers is in the tag-processing portion of `htmlparser`, which scans for pairs of angle brackets and backslashes. This version of `htmlparser` incorrectly handles several angle brackets around the CDATA portion of the file, which causes a mismatch between opening and closing brackets and leads, ultimately, to an exception being thrown. For this input, the only parts that cannot be changed are the angle brackets and backslashes, which are explicitly matched by the tag parser. Again, the portions of the input that are most likely to be sensitive—the contents of the web page and the style sheet—have all been changed.

Overall conclusions. For the three failure-inducing inputs that I presented above, and the additional 167 inputs that I considered in the evaluation, CAMOUFLAGE was always able to anonymize the inputs by removing all of the portions of the inputs that I considered to

be sensitive. These results are encouraging because they provide initial, yet strong evidence that CAMOUFLAGE can generate anonymized failure-inducing inputs that could be safely sent to developers.

4.4.6 RQ4: Improvement

RQ4 investigates the benefits provided by path condition relaxation and breakable input conditions. Ideally, I would have liked to answer RQ4 by comparing CAMOUFLAGE against Castro and colleagues' implementation. Unfortunately, their implementation (1) targets x86 binaries and (2) is not publicly available. I therefore performed a proxy study in which I compared CAMOUFLAGE with a Java implementation of Castro and colleagues' technique that I developed using JPF's symbolic execution engine. Specifically, I compared, for the two tools, the time needed to generate anonymized inputs, the percentage of bits of information revealed by such anonymized inputs, and the percentage of residue between the original and the anonymized inputs. (These measures are the same ones that I used in Sections 4.4.3 and 4.4.4.)

Figure 19 shows the average improvement achieved by CAMOUFLAGE over my implementation of Castro and colleagues' technique, in terms of percentage of bits of information revealed and percentage of residue. As the figure shows, for all of the 20 faults considered, CAMOUFLAGE provides better results. On average, the anonymized inputs generated using CAMOUFLAGE revealed 30% less bits of information and contained 40% less residue. Moreover, the use of path condition relaxation and breakable input constraints increased only marginally the time needed to generate anonymized inputs. With the caveat of a potential implementation bias, these results provide evidence that the use of path condition relaxation and breakable input constraints can substantially improve the basic technique and the overall input-anonymization process.

4.4.7 Threats to Validity

Because I used a limited number of subjects and faults, the results may not generalize. However, both the subjects and the faults I considered are real and representative of the type of situations I expect to encounter in practice. Therefore, I believe that these results,

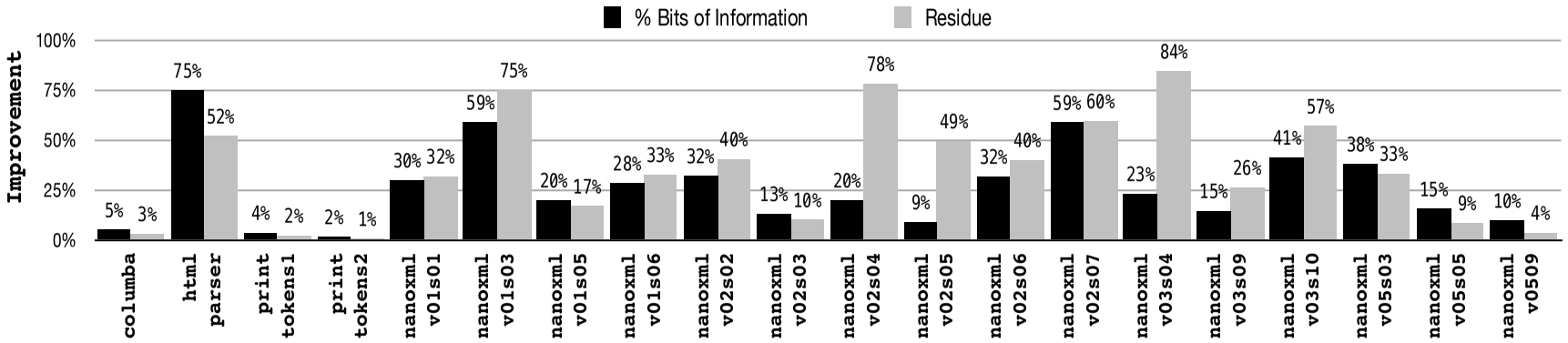

Figure 19: Improvement provided by my approach over the basic approach in terms of bits of information revealed and residue.

albeit preliminary, are promising and motivate further research. An additional threat is that I determined whether the information remaining after anonymization is sensitive, which may introduce bias. Although a human study could have eliminated this threat, I do not think that the effort involved in such a study is justified at this stage of the research. Moreover, as I mentioned earlier and showed in Section 4.4.5, for the cases considered the distinction between sensitive and non-sensitive data was fairly clear-cut.

4.5 Related Work

The technique most closely related to my approach is that by Castro and colleagues [11], which I extended through the novel concepts of path condition relaxation and breakable input conditions. In addition, I perform a more extensive and thorough evaluation on a wider range of inputs and different application types. Finally, the evaluation shows that, for the cases considered, my extensions can considerably improve the effectiveness of input anonymization.

Broadwell and colleagues' `Scrash` tool uses a form of secure information flow (or dynamic tainting) to identify where sensitive information is stored inside a crash dump [7]. During an execution, an initial set of data is marked as sensitive. As the execution progresses, any data that is derived from this initial set is also marked as sensitive. If a crash occurs, any data that is marked as sensitive is excluded from the crash dump that is sent to developers. The main practical limitation of this approach is the difficulty in identifying the initial set of sensitive data—it is unreasonable to expect users to perform this step, and relying on the application's developers is equivalent to trusting them with access to the sensitive data. Furthermore, unlike my technique, `Scrash` does not attempt to anonymize sensitive data, but simply avoids sending it to the developers, which can result in a loss of potentially useful information. In addition, their technique is performed on-line and, unlike my technique, may subject users to high runtime overheads.

Wang and colleagues propose an approach, `Panalyst` [89], that aims to reconstruct failure-inducing inputs on developers' machines by using a combination of dynamic taint analysis, symbolic execution, and collection of answers to questions sent to a client running

on the user's machine. Answers provided by the client determine which direction the symbolic execution takes when encountering branches that depend on sensitive information and what values are read or written by memory accesses through sensitive pointers. The client will answer all questions that do not involve sensitive information, but will only disclose up to a predetermined amount of sensitive information. Besides involving a considerable amount of infrastructure, the main practical limitation of this technique is, like for `Scrash`, the difficulty in identifying which information is sensitive.

In addition to these closely related techniques, there is also a large body of work concerned with anonymizing data sets before they are released to the public (e.g., [22, 84, 94, 105]). These approaches aim to maintain statistical properties of the data (e.g., the age distribution across a population) while preventing users of the data from uniquely identifying a specific record (e.g., the age of a specific individual). Typically, this is accomplished by merging data (e.g., grouping ages 0–18) or by adding random noise to the data. Because the conditions for reproducing a failure are typically very specific, these approaches are not suitable for my scenario.

White-box dynamic test generation and fuzzing techniques (e.g., [9, 29, 73, 86]) are also tangentially related to my approach. Instead of solving path conditions to obtain a new set of inputs to reach a known failure, they iteratively generate and solve new path constraints to explore as many execution paths as possible.

CHAPTER V

IDENTIFYING FAILURE-RELEVANT INPUTS

This Chapter presents my tainting-based technique for automatically identifying subsets of program inputs that are relevant for investigating program failures. The rest of the Chapter is organized as follows: Section 5.1 provides background information on dynamic tainting; Section 5.2 presents a motivating example that is used to illustrate the technique; Sections 5.3 and 5.4 presents the technique and prototype implementation, respectively; and Section 5.5 presents my evaluation of the technique.

5.1 Background

This section presents background information on dynamic tainting that is necessary for explaining my technique. Intuitively, dynamic tainting consists of (1) associating one or more taint marks (tags) with some data values in a program and (2) propagating these marks as data values flow through the program during execution, thus tracking the information flow within the program. As a simple example, consider the code shown in Figure 20a. Assume that the variables a at line 2 and b at line 3 are associated with with taint marks t_a and t_b, respectively. Given this association, after executing this code, variables x, y, and z would be associated with sets of taint marks $\{t_a\}$, $\{t_b\}$, and $\{t_a, t_b\}$, respectively. Taint mark t_a, initially associated with a, is associated with w because a's value is used to compute w. Analogously, t_a is also associated with y because the value of w, which is now tainted with t_a, is used to compute y. The propagation of taint marks for the remaining variables is analogous.

In this example, taint marks are propagated because of *explicit information flow*, that is, direct involvement of tainted data (i.e., a piece of data that is currently associated with a taint mark) in the computation of another value. Explicit information flow can also be

```
   void foo() {
1.   int a, b, w;
2.   int x, y, z;
3.   a = 11;
4.   b = 5;
5.   w = a * 2;
6.   x = b + 1;
7.   y = w + 1;
8.   z = x + y;
   }
```

(a)

```
   void bar(int a) {
1.   int x, y;
2.   if (a > 10) {
3.     x = 1;
4.   }
5.   else {
6.     x = 2;
7.   }
8.   y = 10;
9.   print(x);
10.  print(y);
   }
```

(b)

```
   void baz(int a) {
1.   int x, y;
2.   if(a > 10) {
3.     x = 1;
4.   }

5.   y = 10;
6.   print(x);
7.   print(y);
   }
```

(c)

Figure 20: Example code for illustrating explicit and implicit information flow.

described as propagation that occurs due to data flow or data dependences[1] in the code (e.g., there is a data-flow chain between y and a).

A less intuitive cause of propagation is *implicit information flow*, which refers to situations in which tainted data affects another value indirectly. Whereas explicit information flow is related to data dependencies, implicit information flow is typically due to control dependences[2] in the code. For example, consider the code in Figure 20b and assume that parameter a is associated with taint mark t_a. Although a's value is not involved in the computation of x, it nevertheless affects x's value through a control dependence: The outcome of the predicate at line 3 decides whether line 4 or line 7 will be executed. Therefore, the value of x at the end of the execution is tainted with $\{t_a\}$. Conversely, y would not be tainted because its value does not depend on a's value in any way.

Note that there are even subtler cases of implicit information flow. For example, consider the code in Figure 20c, which is semantically equivalent to the code in Figure 20b but does not contain an `else` branch for the `if` statement at line 2. Assume that for a specific execution, parameter a has the value 2. In this case, the predicate (now at line 2) still affects the value of x and consequently, x should be tainted with $\{t_a\}$. Note that this type of implicit information flow is difficult to identify by simply observing the execution because x is defined before the predicate is computed and is not redefined afterward.

In the remainder of this Chapter, I refer to the propagation of taint marks along data dependences as *data-flow propagation*, the propagation of taint marks along control dependences as *control-flow propagation*, and the propagation along both data and control dependences as *data- and control-flow propagation*.

5.2 Motivating Example

In this section, I provide an example that will be used in the remainder of the Chapter to illustrate my technique. Figure 21 shows the code for the example, which consists of a small program (`fileinfo`) for displaying information about one or more files. The command line

[1] A statement s_2 is data dependent on a statement s_1 if (1) s_2 uses a variable v that is defined in s_1 and (2) there is a definition-clear path with respect to v between s_1 and s_2.

[2] Informally, a statement s_2 is control dependent on a statement s_1 if (1) s_1 contains a predicate, and (2) depending on the outcome of s_1, s_2 may not execute.

arguments to the program are interpreted as an integer flag, which controls the verbosity of the program, and as a list of one or more file names. If the verbose flag is not set, the program prints the name and size of each file passed as an argument and the total size of these files. If the verbose flag is set, the program also prints the first fifty characters of each file.

Program `fileinfo` contains a memory-related fault: the call to `strcat` at line 16 can cause a heap overflow if (1) the verbose flag is set and (2) one of the files being processed has a size that requires more than nine digits to be represented numerically (i.e., the file is one gigabyte or larger). The ten or more digits necessary for representing the file size, plus the fifty-five characters needed to display the preview (fifty for content and five for formatting and end-of-string character) overflow *out*, the sixty-four byte buffer allocated at line 8. Note that, because this failure is a heap overflow, its effect depends on the specific memory layout during a failing execution; it may cause a crash, produce a wrong result, or appear to work correctly while silently corrupting the program's memory.

Although this program and fault are relatively simple, they possess several characteristics that make them a representative example of the type of debugging scenario that the technique targets. The first characteristic is the potential disparity between the amount of code and the input domain. There are less than thirty lines of code, but the amount of inputs (i.e., the number and sizes of the files passed as parameters) can be extremely large. Second, understanding the fault requires tracing interactions among inputs coming from multiple sources. Finally, the number of failure-relevant inputs is likely to be a small percentage of all inputs to the program; regardless of the sizes and number of files passed to `fileinfo`, there are only three failure-relevant inputs: the size and the first fifty characters of the first file larger than one gigabyte and $argv[1]$ (the verbose flag).

5.3 The Technique

This section presents my approach for automatically identifying failure-relevant program inputs. It first provides pseudocode-level view of the algorithm and then provides an intuitive description of the approach and a detailed discussion of its characteristics.

```
  int main(int argc, char **argv) {
1.  int verbose, i, total_size = 0;
2.  struct stat buf;

3.  verbose = atoi(argv[1]);

4.  for(i = 2; i < argc; i++) {
5.    printf("%s: ", argv[i]);

6.    int fd = open(argv[i], O_RDONLY);
7.    fstat(fd, &buf);

8.    char *out = malloc(64);

9.    sprintf(out, "%lld", buf.st_size);

10.   if(verbose) {
11.     char *pview = calloc(55, 1);
12.     strcat(pview, " -[");
13.     int size = read(fd, pview+4, 50);
14.     pview[size+4] = '\0';
15.     strcat(pview, "]");

16.     strcat(out, pview);

17.     free(pview);
18.   }

19.   printf("%s\n", out);
20.   free(out);
21.   total_size += buf.st_size;
22. }

23. printf("total: %d\n", total_size);
  }
```

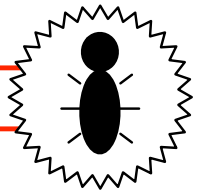

Figure 21: Code for the `fileinfo` utility.

5.3.1 Overview

As mentioned previously, the overall goal of the technique is to help debugging by automatically identifying failure-relevant inputs. The intuition behind the approach is that dynamic tainting, due to its ability to mark and track data at runtime, can be successfully used to accomplish this goal. In this spirit, the approach works by (1) tainting each input read by the application with a unique taint mark, (2) tracking such inputs by suitably propagating taint marks during execution, and (3) checking, when a failure occurs, which taint marks are associated with data involved in the failure. The taint marks identified in the third step allow for separating failure-relevant inputs from inputs that are not relevant for debugging the considered failure.

Algorithm 3 Pseudocode describing how failure-relevant inputs are highlighted.

Input: P: Subject program.
Input: I: Input that causes failure f to manifest in P
Input: C: Relevant context for f
Output: I_f: A subset of I that is relevant for investigating f.

```
1:  procedure HighlightRelevantInputs(P, I)
2:      M = ∅
3:      for i ∈ I do                          ▷ associate a unique taint mark with each input
4:          M = M + [taintMarkFor(i) ↤ i]
5:      end for

6:      T = executeInstrumented(P, I, C, M)   ▷ identify failure-relevant taint marks

7:      for t ∈ T do                          ▷ map from relevant taint marks back to inputs
8:          I_f = I_f ∪ M[t]
9:      end for
10: end procedure
```

Algorithm 3 presents pseudocode describing the algorithm for highlighting subsets of an input that are relevant for investigating a failure. As input, the algorithm takes a subject program P, an input I that causes a failure f to manifest, and a relevant context C.

In the algorithm, Lines 3–5 comprise the first step of the algorithm: associating a taint mark with each input. The second step is performed by line 6 which calls *executeInstrumented*. This function executes an instrumented version of the program that uses the provided relevant context and set of taint marks to determine which taint marks reach data

involved in the failure. Finally, lines 7–9 comprise the third step of the algorithm which constructs I_f by mapping the identified inputs back to their associated inputs.

Before presenting the technical details of each step of the algorithm, I discuss how it would work on the `fileinfo` example from Figure 21. Consider an execution where the program is invoked as "`fileinfo 1 foo.txt bar.txt baz.txt`" and assume that the sizes of the three files passed as arguments are 512 bytes, 1024 bytes, and 1.5 gigabytes, respectively. This execution results in a failure because the verbose flag is set and the size of `baz.txt` is greater than one gigabyte.

Figure 22 provides an intuitive view of how the technique operates in this case. The left-hand side of the figure depicts the inputs to the program (i.e., the command line arguments and the contents and attributes of the files provided as arguments). The name, content, and attributes of each file are shown separately because they are treated as distinct inputs by the program.

The right-hand side of the figure shows the outputs of the program. Because the verbose flag is set, and three files are passed as arguments, there are four outputs. The first three outputs are produced by the calls to `printf` at lines 5 and 19 and comprise the name, size, and first fifty characters (elided in the figure to conserve space) of the corresponding file. The fourth output, produced by the call to `printf` at line 23, shows the cumulative size of the three files. The bug icon in the figure represents the *relevant context* for the failure (i.e., the point in the execution where the failure occurs and the data involved in the failure). For this example, the relevant context consists of the third execution of line 16 and the data in *pview* and *out*.

The lines that traverse the program illustrate, intuitively, how the technique tracks inputs at runtime, by associating unique taint marks to the inputs and propagating such marks as the inputs flow through the program. For example, the lines connecting the attributes of `foo.txt` to Outputs 1 and 4 indicate that there is a flow of information between these inputs and outputs in the execution considered. More specifically, the attributes affect the value of Output 1 because of a data dependence between $buf.st\_size$ (which is part of the attributes) and *out* at line 9; the attributes also affect the value of Output 4 because

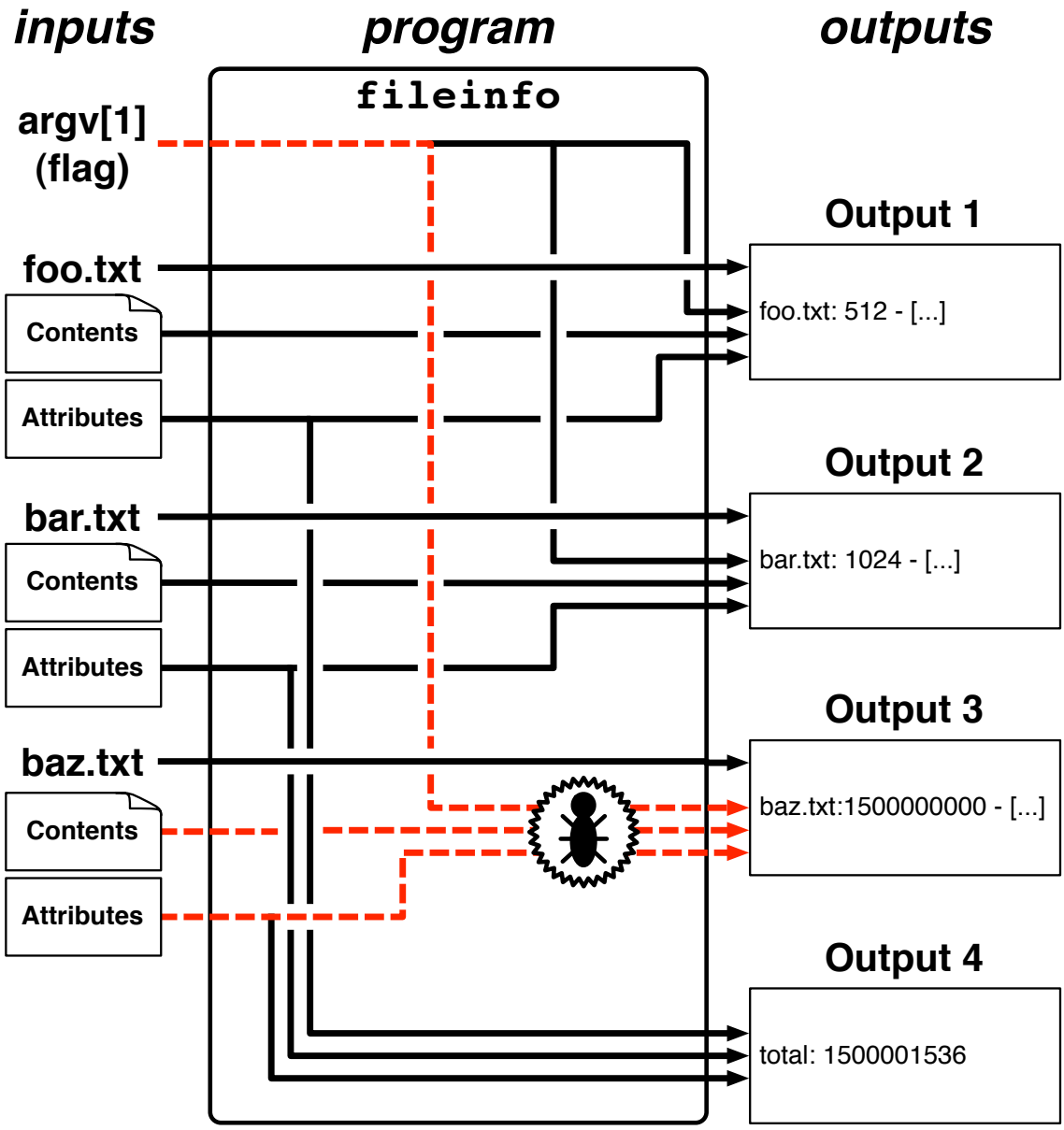

Figure 22: Intuitive view of the application of the technique to an execution of `fileinfo` from Figure 21.

of a data dependence between $buf.st\_size$ (also part of the attributes) and $total\_size$ at line 23.

The technique uses this information to identify which inputs are failure-relevant. When the failure occurs, the technique would determine which taint marks are associated with the data in the relevant context, identify the corresponding inputs, and report such inputs as failure-relevant. In the example, as Figure 22 intuitively depicts, the taint marks associated with the relevant context would map back to $argv[1]$ (the verbose flag) and to contents and attributes of `baz.txt`. Therefore, the technique would report these inputs as failure-relevant to the developers.

In the remainder of this section, I discuss the main technical aspects of the technique by describing in detail its three parts: input tainting, taint propagation, and identification of failure-relevant inputs.

5.3.2 Step 1: Input Tainting

Input tainting is responsible for associating taint marks with inputs as inputs enter an application. To do this, the technique intercepts all interactions between the application and all input sources (e.g., keyboard, mouse, network connections, file systems) and suitably marks data coming from these sources.

The technique assigns taint marks in one of three ways: per-byte, per-entity, or ad-hoc. The *per-byte* mode is the default tainting mode. As indicated by its name, this mode assigns a unique taint mark to each byte of input read by the application. By working at a low level of granularity, the per-byte mode has the potential to perform a fine-grained, and thus precise, identification of failure-relevant inputs. However, most inputs are structured, and applying a unique taint mark to each byte may be unnecessarily expensive in many cases. Consider, for instance, a call to function `gettimeofday`, which reads the system clock. Although multiple bytes are read by this function, they are unlikely to be meaningful individually, whereas together they represent the current time. By treating these bytes as a single input, the technique can reduce the number of taints marks that it needs without losing any precision. (Reducing the number of taint marks is beneficial because it may make

the technique more scalable by speeding up the propagation of taint marks and reducing the amount of space needed to store taint marks.)

To address situations where per-byte tainting would be unnecessarily expensive, the technique provides a *per-entity* tainting mode. When possible, this mode treats related bytes as a single input and taints them using a single taint mark. There are many cases where relationships among individual bytes can be inferred. The bytes that compose individual command line arguments (e.g., $argv[1][0 \ldots n]$, $argv[2][0 \ldots m]$) are likely to represent an atomic piece of information. Analogously, the bytes read by some common library functions (e.g., `stat` or `fstat`), have a well-defined structure that can be leveraged when applying taint marks to them. For such cases, I created specific tainting strategies that apply a single taint mark to groups of related bytes.

The third option provided by the technique, the *ad-hoc* mode, allows developers to define custom taint assignment strategies. This mode is useful when developers' domain knowledge allows them to identify relationships among inputs that would not otherwise be inferred. Consider, for example, a program that reads structured records from a file. Without additional information, the technique would have no way of knowing the relationships among the contents of the file. A developer familiar with the system, however, could easily define a tainting policy that encodes this information and optimize the use of taint marks for such inputs.

The technique performs two additional actions when assigning a taint mark t to an input i. First, it logs a pair (t, src), where src is the source of the input. The source is expressed differently depending on its type (e.g., a fully qualified path name for a file or a network address and port for a network connection). Second, it logs a pair (t, v_i), where v_i is the value of input i. Logging the values of inputs is necessary to handle cases where the input source is transient (e.g., a network connection) and data read from that source is difficult, if not impossible, to recover. The information recorded by these actions is used by the technique when identifying failure-relevant inputs, as described in Section 5.3.4.

As a concrete example of how the tainting part of the technique operates, consider again the code in Figure 21. In this code, there are three locations where inputs are read by the

application: variables *argc* and *argv*, the call to `fstat` at line 7, and the call to `read` at line 13. The technique taints *argc* and *argv* in per-entity mode: the four bytes that compose *argc* are tainted with a single taint mark and all of the bytes in each of the arrays in *argv* are assigned the same taint mark (e.g., $argv[0][0 \ldots n]$ is assigned t_i, $argv[1][0 \ldots m]$ is assigned t_j, etc.). The technique would also operate in per-entity mode for inputs read by `fstat`; by leveraging its knowledge about the structure of the data read by `fstat`, the technique would be able to assign a unique taint mark to each distinct part of the data (e.g., file size, file owner, last access time). Finally, inputs read by the `read` function would be tainted in per-byte mode, with each of the bytes being assigned a unique taint mark.

5.3.3 Step 2: Taint Propagation

The policy for combining taint marks is fairly intuitive. The technique taints all values written by a statement with the union of all taint marks associated with values read by that statement. For instance, after the execution of statement $x = y + z$, where y and z are tainted with taint marks t_1 and t_2, respectively, x would be associated with the set of taint marks $\{t_1, t_2\}$.

When choosing which dependences to consider, the two options are to propagate along only data dependences (i.e., data-flow propagation) or to propagate along both data and control dependences (i.e., data- and control-flow propagation).[3] To assess which option would be appropriate for the technique, I created several small code examples and applied the technique to them. During this preliminary investigation, I was able to construct both cases where data-flow propagation provided better results and cases where data- and control-flow propagation performed better.

Figure 23 shows, intuitively, the relationship between inputs identified using data-flow propagation and inputs identified using data- and control-flow propagation. The outer-most circle represents the set of failure-inducing inputs (i.e., the set of inputs that make the program fail). The union of the two gray areas represents the failure-relevant inputs (i.e., the subset of failure-inducing inputs that are actually relevant for investigating the

[3]Performing control flow propagation alone is not an option, as dependence chains must also involve data dependences.

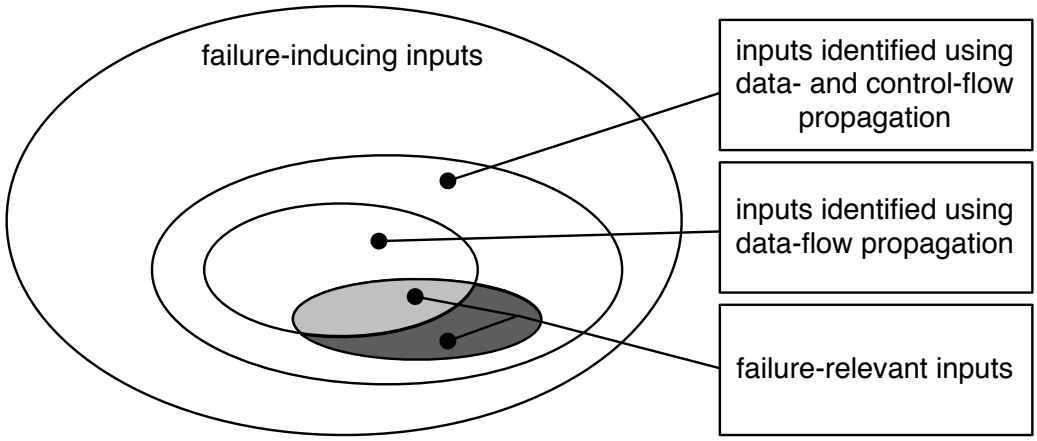

Figure 23: Impact of propagation choice when identifying failure-relevant inputs.

failure), which is the target. When using data- and control-flow propagation, the technique is conservative and identifies all inputs that may *affect* the data involved in the failure. As Figure 23 illustrates, the set of affecting inputs is a superset of the failure-relevant inputs. Therefore, there may be cases in which the set of inputs identified using data- and control-flow propagation may be too large to be useful to developers. Using data-flow propagation alone, the technique would identify smaller-sized sets, but at the cost of incompleteness. For subjects and failures such that the size of the darker gray set is negligible with respect to the size of the lighter gray set, data-flow propagation is likely to be preferable. Conversely, for subjects and failures where the size of the dark gray set dominates the size of the light gray one, data- and control-flow propagation should be used.

In general, there is no way to know *a priori* which propagation option will perform better for a specific program and failure. Therefore, I defined the technique so that it performs both kinds of propagation. The suggested usage scenario for the approach is one where developers start investigating a failure using data-flow propagation first and then, if necessary, use data- and control-flow propagation. As the results of the empirical study in Section 5.5.3 show, this usage scenario is effective for the subjects and faults I considered, and data-flow propagation alone tends to be more useful than data- and control-flow propagation.

5.3.4 Step 3: Identifying Failure-Relevant Inputs

The third part of the technique is responsible for identifying which inputs are failure-relevant. The technique accomplishes this goal by (1) checking to see which taint marks are associated with the data portion of a relevant context and (2) mapping the identified taint marks back to inputs.

In a traditional manual debugging scenario, the starting point of the debugging process is what I defined as the relevant context: information on where the failure occurs and on which data is involved in the failure. The technique requires exactly the same kind of information, encoded in the form of a developer-provided checking function. A *checking function* has two responsibilities. It must identify when the target failure occurs and it must report which data are involved in the failure. In the `fileinfo` example in Figure 21, for instance, a call to a checking function could be inserted before line 16. When the failure occurs, the function would inform the technique that the contents of *pview* and *out* are the data involved in the failure. Additional details about how checking functions are constructed in practice can be found in Section 5.5.2.

Given the relevant context for a specific failure, the technique identifies, when such failure occurs, the taint marks associated with the data involved. The identified taint marks are then mapped to the sources and values of the corresponding inputs. This information is extracted from the logs generated during input tainting (see Section 5.3.2). Inputs identified as failure-relevant are then provided to developers, who can use them to study the cause of the failure under investigation.

5.3.5 Analysis of the Algorithm

Worst-case performance. It is difficult to provide a worst-case execution time analysis of the algorithm as its performance is primarily dependent on the specific execution caused by the failure-inducing input. While Step 1, associating taint marks with inputs, is linear in the number of inputs and Step 3, identifying failure-relevant inputs, is linear in the number of taint marks that reach the relevant context, Step 2 is polynomial in the number of executed instructions (i.e., to propagate the taint marks for an executed instruction

requires the execution of some number of additional instructions). In practice, the number of propagation instructions can vary widely. For example, a simple addition instruction that does not have to propagate any taint marks requires less than 30 instructions while more complicated situations many require several hundred instructions.

The memory usage of the algorithm also depends primarily on the specific execution that is considered and how taint marks propagate to program data. Let T be the set of all taint marks allocated by the algorithm, n be the amount of space needed to store a single taint mark and M be the memory used by the application. The worst-case memory usage of the algorithm is $(T \times n) \times M$. That is, the algorithm needs sufficient space to store a copy of each taint mark for every memory location used by the program.

Although the worst-case runtime and memory usage performance of the algorithm are high they are unlikely to be encountered often in practice and, as the evaluation in Section 5.5 shows, the approach can be used on real applications.

Soundness and precision. In the context of this technique, the ideal (i.e., correct) subset of failure-relevant inputs consists of only those inputs that the developer needs to debug the considered failure; inputs that are irrelevant are not included. In general, identifying this set is impossible because it depends not only on the specific failure and set of failure-inducing inputs but also on the developer and the tools and techniques that they will use to investigate the failure.

My technique is based on the assumption that inputs that lead to a failure through a chain of data-only or data- and control-flow dependencies are useful for debugging such failure. The evaluation in Section 5.5 demonstrates that this assumption holds for the failures I considered. However, there is no guarantee that the algorithm is sound or precise. It may both include irrelevant inputs and fail to include relevant inputs.

5.4 Implementation

PENUMBRA is a prototype tool that implements the technique. The current implementation consists of two components: a trace generator and a trace processor.

The trace generator is built on top of DYTAN, a generic framework for implementing

tainting-based approaches that I developed in previous work [17]. The trace generator leverages DYTAN's built-in functionality for assigning and propagating taint marks to create a trace file that is then processed by the trace processor. The trace generator also supports the summarization of functions by using a short description of how taint marks propagate from the inputs to the outputs of such functions. For example, instead of recording how taint marks propagate along every statement executed inside of the `strcpy` function, the trace generator only needs to record that the outputs of the function should be assigned the same taint marks that are assigned to the inputs. Summarization can greatly reduce the size of the traces that are generated and also reduces the amount of time needed to process the traces. Currently, the trace generator summarizes most of the commonly used functions in the C library.

The trace processor is implemented in Java. It takes as input trace files generated by the trace generator and computes failure-relevant inputs using data-flow propagation and data- and control-flow propagation.

Operating at the binary level has several advantages (e.g., it allows PENUMBRA to handle shared libraries), but it also introduces several challenges. The most serious challenge is the difficulty of obtaining precise control-flow information. For example, in the presence of indirect jumps, it may be impossible to construct an accurate control-flow graph for a binary program. This imprecision can, in turn, negatively impact the accuracy of taint mark propagation; with an inaccurate control-flow graph, PENUMBRA must be conservative when identifying control-dependent regions and, consequently, may over-propagate taint marks. To reduce the impact of this kind of imprecision, I implemented a static analysis proposed by McCamant and Ernst that can help guide propagation along actual, rather than spurious, control dependences [56].

Currently, PENUMBRA ignores GUI inputs because handling them would require a considerable amount of additional implementation effort, while their omission does not prevent us from suitably evaluating the technique.

5.5 Evaluation

Like most debugging aids, a completely realistic evaluation of the technique would require a comprehensive, full-fledged user study. However, at this stage of the research, I do not believe that the costs associated with this type of human study are justified, and I am mainly interested in gathering initial evidence of the usefulness of the approach. To this end, I used PENUMBRA to investigate two research questions:

RQ1: Effectiveness—Can PENUMBRA assist developers in debugging real failures.

RQ2: Comparison—How does PENUMBRA compare with existing input minimization techniques?

To investigate these research questions, I performed two case studies. In the first study, I investigate whether the information provided by PENUMBRA can effectively assist developers in debugging real failures. In the second study, I perform a comparison between the technique and delta debugging along two dimensions: the amount of manual effort needed to use the tools and (an estimate of) the time needed to diagnose and fix a considered failure given the results provided by each tool. The following sections discuss the subjects, and the experimental protocol and results.

5.5.1 Subjects

The ultimate goal of the technique is to direct a developer's attention to a subset of inputs that are useful for debugging a failure. To perform a meaningful evaluation of the technique, I must therefore consider subjects that process a large amount of inputs. (If I considered programs with few inputs, it would be difficult to assess the benefit that the technique can provide.) With this goal in mind, I selected from related work [16, 103] and from the BugBench benchmark suite [53] five applications: `bc` version 1.06, an interpreter for a numeric processing language; `gzip` version 1.2.4 and `ncompress` version 4.2.4, two file compression utilities; `pine` version 4.44, an email and news client; and `squid` version 2.3, a caching proxy. These programs contain real faults (one per subject) and read many inputs from multiple sources. Table 2 shows, for each of the five subjects, its name and version

Table 2: Subject applications.

		Fault location	
Application	KLoC	Function	Line #
bc-1.06	10.5	more_arrays	177
gzip-1.2.4	6.3	get_istat	828
ncompress-4.2.4	1.4	comprexx	896
pine-4.44	239.1	rfc822_cat	260
squid-2.3	69.9	ftpBuildTitleUrl	1024

(*Application*), its size in thousands of lines of code (*KLoC*), and the location of the fault considered in terms of containing function and line number (*Fault location*).

For each subject, I also selected a set of failure-inducing inputs: for `bc` I used the inputs provided with the subject; for `gzip`, `ncompress`, and `squid`, I constructed an appropriate set of inputs using information provided by the authors of BugBench; and for `pine` I used information provided by SecurityFocus.[4] I present additional details on the set of failure-inducing inputs for each subject in Section 5.5.3.

5.5.2 Experimental Protocol and Results

To collect the experimental data for Penumbra, I performed the following steps for each subject and recorded the wall-clock time each step took to complete:

Setup: To setup Penumbra, I ran the subject with its corresponding set of failure-inducing inputs and used traditional debugging techniques to manually identify the failure's relevant context. To avoid biasing the results, I selected the line of code where the considered failure occurs—identified by inspecting the crash dump generated by the failure—and all data read on that line. In general, developers are free to modify the relevant context based on any additional information, intuition, or domain knowledge that they may have regarding the failure being debugged. Note that, with some additional effort, it may be possible to completely automate this step, at least in some cases, by using tools such as Valgrind's MemCheck [74]. In fact, Tucek and colleagues have demonstrated the feasibility of this idea for certain types of failures [81].

[4] `http://www.securityfocus.com/bid/6120/exploit`

Execution: In the execution step, I ran the subject with its corresponding set of failure-inducing inputs and used the trace generation component of PENUMBRA to generate a trace file. To avoid bias, I did not augment the trace generator with any ad-hoc taint assignment strategies (see Section 5.3.2) or custom summarizations (see Section 5.4).

Post processing: Finally, in the post processing step I ran the generated trace file through the trace processing component of PENUMBRA to identify failure-relevant inputs, considering both data-flow propagation and data- and control-flow propagation.

To collect the experimental data for delta debugging I used a similar procedure; I performed the following steps for each subject and recorded the wall clock time that was needed to complete each step.

Setup: Delta debugging's setup involves the construction of an automated oracle that is capable of reliably detecting the failure at hand. Note that creating an automated oracle for delta debugging is considerably more complex than constructing a checking function for PENUMBRA. First of all, the oracle must be robust and general; unlike a checking function, it must be able to identify the failure considered not only for the specific failing execution, but also for all the executions performed by delta debugging on subsets of the failure-inducing inputs. Moreover, because the failures I am considering cause crashes, but do not produce incorrect output, the oracle must take into account the internal state of the subject as the subject executes. I built such oracles by using `gdb` to inspect the stack trace at the failure point and the values of several, manually identified, pieces of program data.

In addition to observing the internal state of the applications, oracles for delta debugging also need to consider how long an execution should take to complete. Because delta debugging removes inputs, there are cases where an execution may not terminate. This is typically the case for interactive applications, such as `bc` and `pine`. To handle this situation, oracles typically use a timeout value and terminate executions that continue for longer than such timeout. For executions that are terminated because of the timeout, the oracle assumes, possibly incorrectly, that inputs used for

the execution did not cause the failure. And if the oracle's assumption is incorrect (i.e., the execution would have failed if it were allowed to continue executing), delta debugging will produce incorrect results. In contrast, the technique does not need to be concerned with execution times since it only requires a single run of the originally-observed failing execution to identify failure-relevant inputs.

Because a timeout that is too short may cause incorrect results, it is important to err of the side of longer values when selecting timeouts. For `bc` and `pine` I found that a 1-second timeout was long enough to prevent incorrect results.

Execution: In delta debugging's execution step I generated a subset of the subject's failure-inducing inputs by running an implementation of delta debugging provided by Zeller[5] on the subject with the subject's corresponding set of failure-inducing inputs and the automated oracle created in the previous step as inputs.

Tables 3 and 4 present the experimental data that I generated using the steps described above. The table is divided by vertical double bars into three sections. In each table, the first two columns show the name and version of each subject (*Application*) and the number of inputs in the subject's set of failure-inducing inputs (*# Inputs*). Note that the number of inputs refers to the number of times PENUMBRA assigned a taint mark to an input. In other words, inputs that are not actually read by the application are not counted. For example, if an application took a ten megabyte file as input but only read the first five bytes of the file, I would consider the number of inputs to be five (rather than ten million). Counting inputs in this way eliminates the risk of obtaining artificially inflated results that favor PENUMBRA.

The third and fourth columns in Table 3 show the number of failure-relevant inputs identified by PENUMBRA when using data-flow propagation (*DF*) and data- and control-flow propagation (*DF & CF*). The final four columns show, in seconds, the wall-clock time measurements for each of the three steps in the experimental protocol for PENUMBRA, described above (*Setup*, *Exec.*, and *Post.*) and the sum of these measurements (*Total*).

[5] `http://www.st.cs.uni-saarland.de/dd/DD.py`

Table 3: Experimental data for PENUMBRA for investigating Study 1 and Study 2.

		# Relevant		Time (s)			
Application	# Inputs	DF	DF & CF	Setup	Exec.	Post.	Total
gzip-1.24	10,000,056	1	3	163	295	9,307	9,765
ncompress-4.2.4	10,000,056	1	3	70	185	6,053	6,308
pine-4.44	15,103,766	26	15,100,344	97	80	2,117	2,294
bc-1.06	1373	209	743	314	4	4	322
squid-2.3	1,402,056	89	2,056	125	137	212	443

Table 4: Experimental data for delta debugging for investigating Study 1 and Study 2.

			Time (s)		
Application	# Inputs	# Remaining	Setup	Exec.	Total
gzip-1.24	10,000,056	1	1,800	129	1,929
ncompress-4.2.4	10,000,056	1	1,800	137	1,937
pine-4.44	15,103,766	90	5,400	7,036	12,436
bc-1.06	1373	285	12,600	1,727	14,327
squid-2.3	1,402,056	—	—	—	—

The third column in Table 4 shows the number of inputs included in the minimized subset of failure-inducing inputs produced by delta debugging (*# Remaining*) and the next three columns show, in seconds, the wall-clock time measurements for the two steps in delta debugging's experimental protocol (*Setup* and *Exec.*) and the sum of these measurements (*Total*). Note that the time measurements for the setup step were recorded in half hour intervals. The dashed lines in the last row indicate that, after a day of work, I was unable to construct an automated oracle for the failure in `squid`. Constructing an oracle for the failure in `squid` is particularly difficult because `squid` is multithreaded, the failure involves interactions between multiple processes, and a considerable amount of infrastructure is necessary for controlling the environment in which `squid` executes (e.g., redirecting `squid`'s network accesses to the minimized versions of web pages and data generated by delta debugging).

5.5.3 RQ1: Effectiveness for Debugging

The goal of the empirical study that addresses the first research questions is to investigate whether the information provided by PENUMBRA is effective in assisting developers when

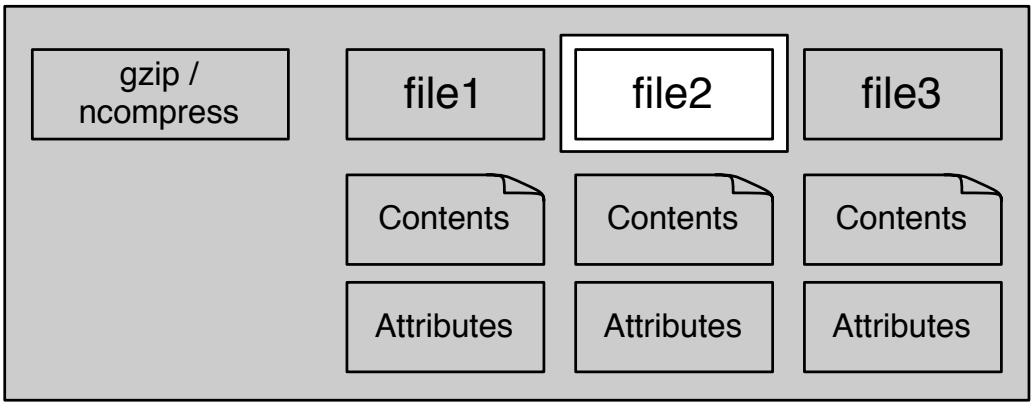

Figure 24: Failure-relevant input for `gzip` and `ncompress` identified using data-flow propagation.

debugging real failures. For each subject, I discuss the information generated by PENUMBRA and provide a qualitative assessment of its usefulness to developers. In the discussion, I assume that developers are following the scenario I presented in Section 5.3.3: first they consider failure-relevant inputs identified using data-flow propagation and then, if necessary, consider failure-relevant inputs identified using data- and control-flow propagation.

Gzip and Ncompress: I present these two subjects together because they are very similar. They are both file compression utilities and their faults are nearly identical: each fault causes a buffer overflow if a file with a path length greater than 1024 characters is compressed. Therefore, I used the same set of failure-inducing inputs for both subjects.

Figure 24 shows a representation of the inputs used to cause a failure in `gzip` and `ncompress`. Each program is given three files to compress: `file1`, `file2`, and `file3`, where `file2` has a path length greater than 1024. As shown in Table 3, the total number of failure-inducing inputs is 10,000,056. The majority of these inputs (10,000,000) are read from the contents of `file1`. The remainder comes from several sources (e.g., `argc`, `argv`, and `file1`'s attributes). Note that the contents and attributes of `file2` and `file3` are not counted as failure-inducing inputs because they are not read before the applications fail.

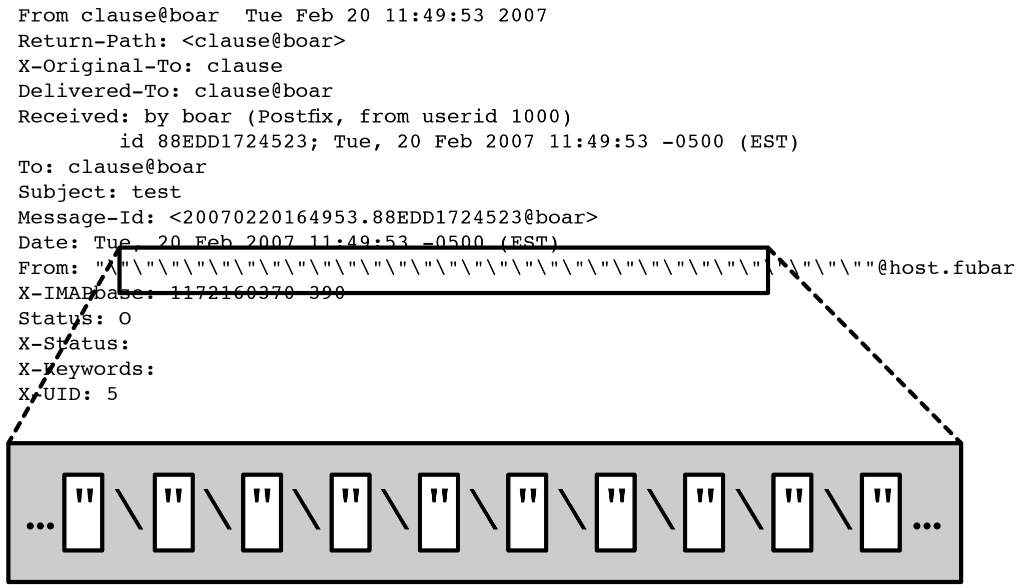

Figure 25: Excerpt of failure-relevant input for `pine` identified using data-flow propagation.

Using data-flow propagation, Penumbra identifies `file2`'s path as the only failure-relevant input. In Figure 24, the single failure-relevant input is shown inside of a white box. This is an encouraging result because the information provided by Penumbra correctly suggests two conclusions: (1) that `file2`'s path is involved in the failure and (2) that the rest of the failure-inducing inputs are not involved in the failure. These conclusions closely match the actual failure conditions and are likely to direct developers right at the core of the problem.

For this failure, using data- and control-flow propagation would lead to only a minimal increase in the number of inputs identified as failure-relevant. Only two additional inputs would be identified and reported by Penumbra: the name of the executable and the path of `file1`. The contents of `file1` would still not be reported, and developers would still be presented with information that is likely to point them directly to the problem.

Pine: The failure in `pine` is triggered when the program processes a mailbox containing an email with a “from” field that contains 22 or more double quotes. The failure-inducing inputs I selected include a mailbox with a size of approximately 15 megabytes that contains an email with a suitably-crafted “from” field. Figure 25 shows an excerpt of

such a problematic email message.

Using data-flow propagation, PENUMBRA identifies 26 inputs as failure-relevant: the 26 double quotes in the problematic "from" field. The zoomed view in Figure 25 shows these inputs inside white boxes. This is also an encouraging result. Like in the cases of `gzip` and `ncompress`, the information provided by PENUMBRA using data-flow propagation correctly highlights a subset of the failure-inducing inputs that is necessary for the failure to manifest.

Differently from the failures in `gzip` and `ncompress`, for this failure, data- and control-flow propagation identifies many additional inputs, nearly 15 million—almost the whole mailbox. This is an example of a situation where a large number of inputs potentially affect the data involved in a failure, but relatively few are actually relevant for investigating the failure. On the positive side, the relevant inputs identified through data-flow propagation for `pine` are likely to be sufficient for diagnosing the failure, so the developer would not need to consider this second set of inputs.

Bc: The fault in `bc` manifests when the program processes a script that (1) allocates at least 32 arrays and (2) has allocated more variables than arrays when the 32^{nd} array is created. Figure 26 shows the relevant portions of the problematic script. In the language that `bc` processes, variables can be allocated using the `auto` keyword (i.e., `auto a1, a2`, etc.), and arrays are allocated when they are referenced (e.g., `p1[1], p2[1]`).

Using data-flow propagation, PENUMBRA identifies 209 inputs as failure-relevant. These are the inputs shown inside the white boxes in Figure 26 and correspond to the names of variables and arrays in the script. The set of failure-relevant inputs only contains the names of 31 arrays, which is one short of the 32 arrays that are needed to trigger the failure. However, the identified inputs strongly suggest that the considered failure involves interactions between the processing of variables and arrays, so I believe this result is still promising.

Using data- and control-flow propagation for this failure causes an additional 534

```
/* An example that finds all primes between 2 and limit. */

define tests (limit) {

    auto a1, a2, a3, ..., a62, a63, a64

    p1[1]   =  1;
    p2[1]   =  1;
    p3[1]   =  1;
    p4[1]   =  1;
    p5[1]   =  1;
    ...
    p29[1]  =  1;
    p30[1]  =  1;
    p31[1]  =  1;
    p32[1]  =  1;
    p33[1]  =  1;
    p34[1]  =  1;
    ...
    p68[1]  =  1;
    p69[1]  =  1;
}

print "\ntyping 'tests (10)' will construct 10 elements array.\n"
quit
```

Figure 26: Excerpt of failure-relevant input for `bc` identified using data-flow propagation.

inputs (743 total) to be identified as failure relevant. Although this is still a subset of the total inputs, the large number of inputs identified as failure-relevant is likely to obscure the actual cause of the failure.

Squid: The fault in `squid` manifests when the program processes an FTP request where the username, password, and host name components of the request combined contain more than 64 characters, which rfc1738[6] defines as unsafe. The failure-inducing inputs I selected include a sequence of 50 HTTP and FTP requests, drawn from the my daily internet usage, that fetch data and webpages totaling approximately 1.5 megabytes in size. The sequence of requests also contains a suitably crafted FTP request that causes the failure. Figure 27 shows an excerpt of the request sequence that includes the problematic FTP request, in addition to several other HTTP requests.

[6] `http://www.w3.org/Addressing/rfc1738.txt`

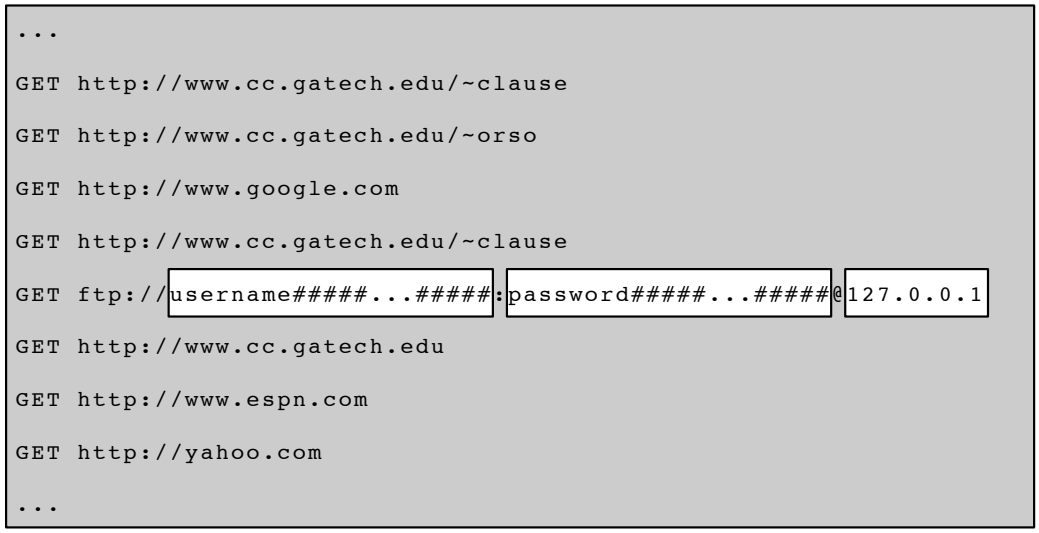

Figure 27: Excerpt of failure-relevant input for `squid` identified using data-flow propagation.

Using data-flow propagation, PENUMBRA identifies 89 inputs as failure-relevant. These inputs are shown in Figure 27 surrounded by white boxes. Like for the other failures I have investigated, PENUMBRA's performance using data-flow propagation is encouraging. The failure-relevant inputs suggest that the username, password, and host name components of the problematic FTP request are involved in the failure, whereas the webpages and data read from the network are not involved in the failure.

Using data- and control-flow propagation, an additional 1,967 inputs are identified as failure-relevant (2,056 total). These additional inputs correspond to the majority of the requests, but the contents of the webpages and FTP data are still completely excluded. Therefore, even when considering this larger subset, there would still be a significant reduction in the amount of data developers would need to examine when investigating the failure.

Based on the results of this first empirical study, I can make some initial observations about the effectiveness of the information provided by the technique. For all of the five subjects considered, PENUMBRA computed failure-relevant inputs that closely correspond to the necessary conditions for reproducing the considered failures. Moreover, for these failures, only data-flow propagation was needed to provide useful information. Additionally, for the considered failures in `gzip`, `ncompress`, and `squid`, data- and control-flow propagation also

provided results that are likely to be useful to developers. Conversely, for `bc` and `pine`, data- and control-flow propagation seems to identify too many inputs for the information to be useful. However, because data-flow propagation provides useful information in all of the cases I examined, according to the scenario of usage, developers would be unlikely to consider the use of data- and control-flow propagation. Therefore, I believe that the results of this case study, albeit preliminary, suggest that the technique can be effective in assisting debugging of failures in real applications.

5.5.4 RQ2: Comparison with Delta Debugging

The goal of the study that addresses the second research question is to provide a quantitative comparison between PENUMBRA and delta debugging. In my experience, there are two main dimensions that developers consider when comparing debugging tools: (1) the amount of manual effort involved in using the tools and (2) the amount of time needed to diagnose and fix a considered failure given the information provided by the tools.

As a proxy for the amount of manual effort needed to use the tools, I used the amount of time needed to complete their setup steps. For both tools, the other steps can be fully-automated and performed off-line (e.g., overnight, or as part of an automated build system). Therefore, the time necessary for completing these steps is of little relevance, as long as it is of the same magnitude for both tools (which is the case for PENUMBRA and delta debugging).

Figure 28 shows, for `gzip`, `ncompress`, `bc`, and `pine`, a comparison between the amount of setup time needed to use PENUMBRA and the amount of setup time needed to use delta debugging. (These numbers are the same as the ones in Tables 3 and 4 and are repeated here for the convenience of the reader.) As the figure shows, for these four subjects the amount of setup time needed for PENUMBRA is far less than the amount needed for delta debugging. This large difference in the amount of setup time between the two tools is likely to make PENUMBRA more practical and to be a strong incentive for its use by developers.

As an estimate of the relative amounts of time needed to diagnose and fix failures, given the information provided by the tools, I compared the number of inputs that each tool identifies as relevant.

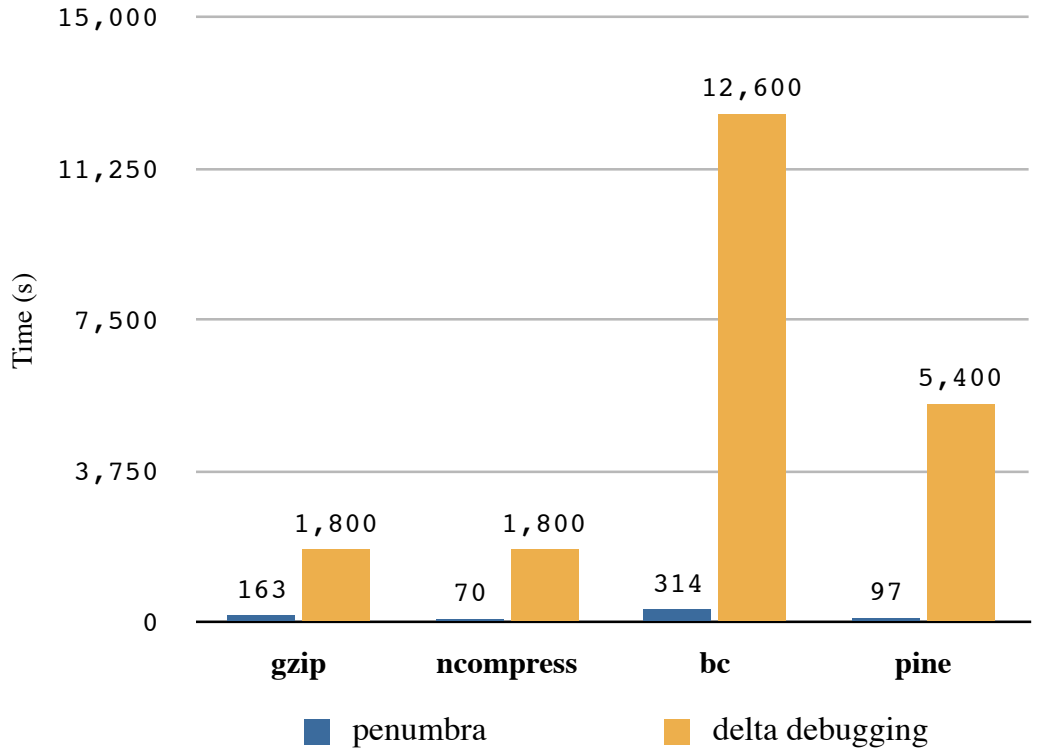

Figure 28: Comparison between the amount of setup time needed for PENUMBRA and delta debugging.

Tables 3 and 4 shows that, for the failures I considered, the number of inputs that PENUMBRA identifies using data-flow propagation is relatively close in size to the number of inputs identified by delta debugging. In addition, I found that the inputs identified by delta debugging are a superset of the inputs identified by PENUMBRA using data-flow propagation. If I consider the amount of relevant inputs for a failure to be an indicator of the amount of developers' time required to diagnose such failure, I expect that the two tools will have comparable performance, with PENUMBRA being slightly more effective, at least in some cases.

Table 3 also shows that for two subjects, `bc` and `pine`, the number of failure-relevant inputs identified by PENUMBRA using data- and control-flow propagation is significantly larger than the number of inputs identified by delta debugging. This means that for the failures in these subjects, the amount of time needed to diagnose and fix the considered failure using information provided by delta debugging will likely be less than the amount of

time needed when using information provided by PENUMBRA using data- and control-flow propagation.

One final point to consider in this comparison is the reduction in the number of program statements that developers need to consider when investigating a failure that each tool provides. The set of minimized inputs that delta debugging provides corresponds to a minimized execution that still produces the considered failure and that can be readily used by the developer. Conversely, my technique identifies failure-relevant inputs, but does not automatically produce a minimized execution. Although it would be straightforward to compute a dynamic slice starting from such inputs (PENUMBRA itself could be easily modified to produce such a slice), computing an actual minimized execution would require some less straightforward extensions to my technique.

Based on the results the second case study, I believe that PENUMBRA compares favorably with delta debugging and provides several incentives for its use in practice. First, for the subjects and failures I considered, PENUMBRA needed considerably less setup time than delta debugging, which suggests that PENUMBRA will also require less manual effort. Second, the number of failure-relevant inputs identified by PENUMBRA using data-flow propagation is comparable in size to the number of inputs identified using delta debugging. This observation, combined with the results from the first case study (which suggest that data-flow propagation may be sufficient to produce useful results in most cases), indicates that PENUMBRA and delta debugging are likely to provide a similar level of effectiveness for developers who are diagnosing and fixing failures.

Finally, note that, although both tools share the same goal, they are not mutually exclusive. In practice, delta debugging could be used in conjunction with PENUMBRA by slightly extending the usage scenario I proposed in Section 5.3.3, as follows. Developers could first use PENUMBRA, which requires a smaller amount of setup time, to investigate a failure. If the information provided by PENUMBRA, using data-flow propagation first and data- and control-flow propagation later, is sufficient for diagnosing the failure, developers would terminate their debugging activity. Otherwise, they could invest the additional time needed to setup and run delta debugging and use the minimized input that it provides to

further investigate the considered failure.

5.5.5 Threats to validity

The fact that I performed the steps necessary for generating the experimental data is clearly a threat to the external validity of the experiments, and the conclusions that are drawn from it may not generalize. These threats are mitigated by the fact that I am equally skilled with both techniques, as I used delta debugging extensively in previous work [18]. Therefore, even if the amount of time necessary for each step changes when they are performed by other developers, it is likely that the ratio between them will remain fairly constant. Another threat to the external validity of the study is that I used a limited number of subjects and faults. However, both the subjects and the faults I considered are real, and the debugging scenario investigated is representative of a real scenario.

5.6 Related Work

Because of the challenges and importance of debugging, there is large amount of work that is related to my technique. For the sake of space, I present only the most closely related of such approaches.

Given a failure and a set of failure-inducing inputs, delta debugging runs the failing program multiple times, each time with a different subsets of the inputs, in the attempt to find a minimized subset that still causes the program to fail [100]. As I mentioned previously, delta debugging is an effective technique, but its practical usefulness can be limited by the need for a precise oracle and by the time required to rerun a failing program many times. Mishergi and Su present an extension to delta debugging that can reduce the number of executions needed when operating on well-defined, tree-structured inputs [60], but currently no solution exists for the general case.

Chan and Lakhotia also propose several methods for identifying failure-relevant inputs [12]. The one most closely related to my technique uses dynamic slicing (described in the next paragraph) to trace incorrect elements in a program's output back to statements that read inputs that affect the incorrect output elements. Given these statements, developers must then identify which inputs are read by the statements. Compared to my

technique, this approach requires much more manual interaction by developers, and as their evaluation shows, the amount of work involved becomes prohibitive on subjects as small as 150 lines of code.

The rest of the approaches that I discuss in this section are not alternatives to my technique, but rather complementary—they are code-centric approaches that could be combined with my technique to increase the overall effectiveness of the debugging effort.

The first family of complementary techniques is based on slicing. Slicing is an analysis technique that uses data and control dependences in a program to locate a set of statements called a *slice*. The set of statements in a (backward) slice may influence the value of a variable at a particular program point. Weiser first proposed the use of *static slicing* for debugging [90]. Static slicing considers static data and control dependences for all possible executions and is often overly conservative. To address this problem, Korel and Laski proposed *dynamic slicing*, in which only dependences that were exercised during an execution are considered when computing slices [47]. More recently, Gupta and colleagues investigated the combination of delta debugging and dynamic slicing [32] to further reduce the size of slices, while Zhang and colleagues combined check-pointing with dynamic slicing for analyzing long-running applications [103].

A second family of complementary approaches attempts to automate debugging by ranking program entities. These approaches include techniques that aim to locate suspicious statements by comparing passing and failing executions. Agrawal and colleagues propose to use the set difference between statements executed in a failing run and statements executed in one or more passing runs as an indication of suspiciousness [1]. Renieris and Reiss improve on this approach by using a nearest neighbor calculation to find the passing execution that is most similar to the failing execution [67]. The closest passing execution is then used to calculate the set difference with the failing execution. Jones and colleagues propose a slightly different approach. Instead of using set difference, they estimate the suspiciousness of a statement by computing a ratio of its appearance in passing and failing executions [42]. Finally, Liblit and colleagues use a statistical approach that correlates predicates with failures [49].

CHAPTER VI

CONCLUSION

The overall goal of my research is to improve software quality. My dissertation work addresses this goal by developing new program analysis techniques that have the potential to help developers debug program failures. My thesis statement is:

Program analysis techniques can enable and support the debugging of failures in widely-used applications by (1) capturing, replaying, and, as much as possible, anonymizing failing executions and (2) highlighting subsets of failure-inducing inputs that are likely to be helpful for debugging such failures.

The evaluation of my thesis statement consisted of two parts. In the first part, I designed and implemented a set of techniques that demonstrate that the hypothesized capabilities of program analysis are possible. In the second part, I demonstrated that the techniques I developed can be applied to medium- to large-sized, widely-used applications and that they can enable and support the debugging of real faults in such applications.

6.1 Merit

This dissertation provides several technical and conceptual contributions to the field of software engineering. As mentioned in the Introduction, the most significant are:

- A technique that can enable and support the debugging of failures that are difficult to reproduce by providing developers with a focused test case that recreates an observed failure.
- A technique that can enable the debugging of failures that involve sensitive or privileged inputs by, as much as possible, automatically anonymizing such inputs.

- A technique that supports manual debugging efforts by automatically identifying subsets of a program's inputs that are relevant for investigating a failure.

- Evaluations of the above techniques using real failures in widely-used applications that demonstrate that the techniques are feasible and useful.

In addition to the contributions listed above, my dissertation work has also produced several other tools and techniques that have been, or can be, used by the community.

First, I developed a novel mechanism for recording and storing failing program executions (see Section 3.2.1). This mechanism is vital for my work because it greatly simplifies the minimization process, but it can also be leveraged by other techniques that use captured program executions. In fact, it has been specifically referenced as a potential improvement to their techniques by Ko and Myers [45] and Wang and colleagues [89]. Second, I developed an algorithm for automatically minimizing captured failing executions (see Section 3.2.4). Again, this algorithm is necessary for my work, but it could also easily be repurposed for other goals. For example, Jorde and colleagues [43] briefly describe how the minimization algorithm could be modified to derive unit tests from system tests (i.e., captured failing executions). And finally, as a prerequisite to developing my technique for identifying subsets of failure-relevant inputs, I built and made available a generic taint analysis framework called Dytan [17]. Dytan has been requested by other researchers, both academic and industrial, and has been used to implement a variety of other analyses [16, 19, 39].

6.2 Future Work

In the future, it is likely that debugging will continue to be a major component of the software development processes. By itself, the work in this dissertation helps to address the cost and difficulty of debugging, but it also motivates and lays out a foundation for other debugging tools and techniques.

My techniques for recording, minimizing, and replaying executions, and anonymizing failure-inducing inputs provide developers and researchers with the ability to safely reproduce executions that occurred on a remote machine. I have demonstrated how this ability

can be used to help the debugging of captured failures, but the ability to reproduce executions can also be leveraged in other ways. For example, the techniques could also be used to leverage remote executions do not contain failures. One possibility is to minimize passing executions with the goal of creating test cases that exercise functionality that was not tested in-house. The minimized executions could then be added to an existing test suite to augment the test suite's coverage or to ensure that existing functionality continues to work after changes are made to the software. Another possibility is tweaking (fuzzing) passing executions to transform them into failing executions. By creating failing executions that are similar to passing executions generated by users, it is possible to reveal latent and hard to reach faults that are likely to be encountered by users.

My technique for highlighting subsets of failure-relevant inputs uses data and control dependencies to identify which inputs are related to a failure. This approach works well, but it can be improved. For example, in several cases, using control-dependencies leads to a large number of inputs being identified as failure-relevant. The development of filtering or ranking mechanisms, perhaps based on execution time or the influence of specific branches, could improve the approach. Another possibility is the creation of failure-relevant input identification techniques that use some other mechanism for measuring influence (e.g., information flow between variables). Such approaches may be more suitable for specific application domains.

Finally, it would also be interesting to explore the effects of combining my minimization approach with other techniques. For example, the anonymization technique could be improved by combining the two techniques. As Section 4.4.5 demonstrated, in many cases, the structural components of an input can not be changed. Although these inputs are generally safe to send to developers, a manual inspection is often needed to verify that only structural information remains. If an input is first minimized, then it is possible that much of the structural information can automatically be removed. A similar benefit is also possible if minimization is used with failure-relevant input identification; removing portions of the input can help limit unwanted input identification via control dependencies.

REFERENCES

[1] Agrawal, H., Horgan, J., London, S., and Wong, W., "Fault localization using execution slices and dataflow tests," in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pp. 143–151, 1995.

[2] Apple Inc., "Apple Customer Privacy Policy," 2011. `http://www.apple.com/legal/privacy/`.

[3] Apple, Inc., "Technical Note TN2123: CrashReporter," 2011. `http://developer.apple.com/technotes/tn2004/tn2123.html`.

[4] Automation Anywhere, Inc., "Automated testing software for automating web testing, software testing & more.," 2011. `http://www.automationanywhere.com/Testing/products/automated-testing-anywhere.htm`.

[5] Ball, T., Naik, M., and Rajamani, S. K., "From symptom to cause: localizing errors in counterexample traces," in *In Principles of Programming Languages*, pp. 97–105, 2003.

[6] Bird, C., Nagappan, N., Devanbu, P., Gall, H., and Murphy, B., "Does distributed development affect software quality? An empirical case study of Windows Vista," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 518–528, 2009.

[7] Broadwell, P., Harren, M., and Sastry, N., "Scrash: A system for generating secure crash information," in *Proceedings of the 12th Conference on USENIX Security Symposium*, pp. 19–19, 2003.

[8] Burger, M. and Zeller, A., "Replaying and isolating failing multi-object interactions," in *Proceedings of the 2008 International Workshop on Dynamic Analysis*, pp. 71–77, 2008.

[9] Cadar, C., Dunbar, D., and Engler, D. R., "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 209–224, 2008.

[10] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R., "EXE: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pp. 322–335, 2006.

[11] Castro, M., Costa, M., and Martin, J.-P., "Better bug reporting with better privacy," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 319–328, 2008.

[12] Chan, T. W. and Lakhotia, A., "Debugging program failure exhibited by voluminous data," *Journal of Software Maintenance*, vol. 10, no. 2, pp. 111–150, 1998.

[13] Chang, R. Y., Podgurski, A., and Yang, J., "Finding what's not there: A new approach to revealing neglected conditions in software," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 163–173, 2007.

[14] Chen, S.-K., Fuchs, W. K., and Chung, J.-Y., "Reversible debugging using program instrumentation," *IEEE Transactions on Software Engineering*, vol. 27, no. 8, pp. 715–727, 2001.

[15] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K., "HOLMES: Effective statistical debugging via efficient path profiling," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 34–44, 2009.

[16] Clause, J., Doudalis, I., Orso, A., and Prvulovic, M., "Effective memory protection using dynamic tainting," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 284–292, 2007.

[17] Clause, J., Li, W., and Orso, A., "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 196–206, 2007.

[18] Clause, J. and Orso, A., "A technique for enabling and supporting debugging of field failures," in *Proceedings of the 29th IEEE and ACM SIGSOFT International Conference on Software Engineering*, pp. 261–270, 2007.

[19] Clause, J. and Orso, A., "LEAKPOINT: Pinpointing the causes of memory leaks," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 515–524, 2010.

[20] Cleve, H. and Zeller, A., "Finding failure causes through automated testing," in *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000.

[21] DeMillo, R. A., Pan, H., and Spafford, E. H., "Critical slicing for software fault localization," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 121–134, 1996.

[22] Du, W. and Atallah, M. J., "Privacy-preserving cooperative statistical analysis," in *Proceedings of the 17th Annual Computer Security Applications Conference*, 2001.

[23] Dutertre, B. and de Moura, L., "Integrating simplex with DPLL(T)," Tech. Rep. SRI-CSL-06-01, SRI International, 2006.

[24] Dutertre, B. and de Moura, L., "The YICES SMT solver." `http://yices.csl.sri.com/tool-paper.pdf`, 2011.

[25] Elbaum, S., Chin, H. N., Dwyer, M., and Dokulil, J., "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 241–252, 2006.

[26] Elbaum, S. and Diep, M., "Profiling deployed software: Assessing strategies and testing opportunities," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 312–327, 2005.

[27] Elkarablieh, B., Godefroid, P., and Levin, M. Y., "Precise pointer reasoning for dynamic test generation," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pp. 129–140, 2009.

[28] Georges, A., Christiaens, M., Ronsse, M., and Bosschere, K. D., "JaRec: A portable record/replay environment for multi-threaded Java applications," *Software Practice and Experience*, vol. 34, no. 6, pp. 523–547, 2004.

[29] Godefroid, P., Klarlund, N., and Sen, K., "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 213–223, 2005.

[30] Groce, A., Kroening, D., and Lerda, F., "Understanding counterexamples with `explain`," in *In Computer-Aided Verification*, pp. 453–456, 2004.

[31] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B., "Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 495–504, 2010.

[32] Gupta, N., He, H., Zhang, X., and Gupta, R., "Locating faulty code using failure-inducing chops," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 263–272, 2005.

[33] Gyimothy, T., Beszedes, A., and Forgacs, I., "An efficient relevant slicing method for debugging," in *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 303–321, 1999.

[34] Hao, D., Pan, Y., Zhang, L., Zhao, W., Mei, H., and Sun, J., "A similarity-aware approach to testing based fault localization," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 291–294, 2005.

[35] Hewlett-Packard Development Company, L.P., "Hp functional testing software," 2011. `https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-15-24%5E1322_4000_100__`.

[36] Hilbert, D. M. and Redmiles, D. F., "An approach to large-scale collection of application usage data over the Internet," in *Proceedings of the 20th International Conference on Software Engineering*, pp. 136–145, 1998.

[37] Hilbert, D. M. and Redmiles, D. F., "Extracting usability information from user interface events," *ACM Computing Surveys*, vol. 32, pp. 384–421, December 2000.

[38] IEEE, "IEEE standard glossary of software engineering terminology," Tech. Rep. Std 610.12-1990, IEEE, 1990.

[39] Iozzo, V., "0-knowledge fuzzing," Black Hat DC, 2010. `http://www.cert.org/vuls/discovery/downloads/0knowfuzz-bh.pdf`.

[40] Jiang, L. and Su, Z., "Context-aware statistical debugging: from bug predictors to faulty control flow paths," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 184–193, 2007.

[41] Jones, J. A., Bowring, J. F., and Harrold, M. J., "Debugging in parallel," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 16–26, 2007.

[42] Jones, J. A., Harrold, M. J., and Stasko, J., "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, pp. 467–477, 2002.

[43] Jorde, M., Elbaum, S., and Dwyer, M. B., "Increasing test granularity by aggregating unit tests," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 9–18, 2008.

[44] King, S. T., Dunlap, G. W., and Chen, P. M., "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the USENIX Annual Technical Conference*, pp. 1–15, 2005.

[45] Ko, A. J. and Myers, B. A., "Debugging Reinvented: Asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 301–310, 2008.

[46] Konuru, R., Srinivasan, H., and Choi, J.-D., "Deterministic replay of distributed Java applications," in *Proceedings of the 14th IEEE International Parallel & Distributed Processing Symposium*, pp. 219–228, 2000.

[47] Korel, B. and Laski, J., "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[48] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I., "Bug isolation via remote program sampling," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 141–154, 2003.

[49] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I., "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15–26, 2005.

[50] Liblit, B., *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2004.

[51] Liu, C. and Han, J., "Failure proximity: A fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 46–56, 2006.

[52] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P., "SOBER: Statistical model-based bug localization," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 286–295, 2005.

[53] Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., and Zhou, Y., "Bugbench: Benchmarks for evaulating bug detection tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[54] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K., "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 190–200, 2005.

[55] Martin, J.-P., "Upper and lower bounds on the number of solutions," Tech. Rep. MSR-TR-2007-164, Microsoft Research, 2007.

[56] McCamant, S. and Ernst, M. D., "Quantitative information flow as network flow capacity," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 193–205, 2008.

[57] Microsoft Corporation, "Microsoft customer experience improvement program," 2011. `http://www.microsoft.com/products/ceip/`.

[58] Microsoft Corporation, "Privacy Statement for the Microsoft Error Reporting Service," 2011. `http://oca.microsoft.com/en/dcp20.asp`.

[59] Microsoft Corporation, "Windows Error Reporting: Getting Started," 2011. `http://www.microsoft.com/whdc/maintain/StartWER.mspx`.

[60] Misherghi, G. and Su, Z., "Hdd: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 142–151, 2006.

[61] Narayanasamy, S., Pokam, G., and Calder, B., "BugNet: Continuously recording program execution for deterministic replay debugging," in *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pp. 284–295, 2005.

[62] Netzer, R. H. B. and Weaver, M. H., "Optimal tracing and incremental reexecution for debugging long-running programs," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pp. 313–325, 1994.

[63] Orso, A., Joshi, S., Burger, M., and Zeller, A., "Isolating relevant component interactions with JINSI," in *Proceedings of the Fourth International ICSE Workshop on Dynamic Analysis*, pp. 3–9, 2006.

[64] Orso, A. and Kennedy, B., "Selective capture and replay of program executions," in *Proceedings of the Third International ICSE Workshop on Dynamic Analysis*, pp. 29–35, 2005.

[65] Osterweil, L. J. and Clarke, L. A., "Continuous self-evaluation for the self-improvement of software," in *Proceedings of the First International Workshop on Self-adaptive Software*, pp. 27–39, 2000.

[66] Pavlopoulou, C. and Young, M., "Residual test coverage monitoring," in *Proceedings of the 21st International Conference on Software Engineering*, pp. 277–284, 1999.

[67] Renieris, M. and Reiss, S., "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 30–39, 2003.

[68] Reps, T., Ball, T., Das, M., and Larus, J., "The use of program profiling for software maintenance with applications to the year 2000 problem," pp. 432–449, 1997.

[69] Richardson, D., "Perpetual testing," 2011. `http://www.ics.uci.edu/~djr/edcs/PerpTest.html`.

[70] Russinovich, M. and Cogswell, B., "Replay for concurrent non-deterministic shared-memory applications," in *Proceedings of ACM SIGPLAN Conference on Programming Languages and Implementation*, pp. 258–266, 1996.

[71] Saff, D., Artzi, S., Perkins, J. H., and Ernst, M. D., "Automatic test factoring for Java," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 114–123, 2005.

[72] SeleniumHQ, "Selenium web application testing system," 2011. `http://seleniumhq.org/`.

[73] Sen, K., Marinov, D., and Agha, G., "CUTE: A concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 263–272, 2005.

[74] Seward, J. and Nethercote, N., "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the USENIX Annual Technical Conference*, 2005.

[75] Srinivasan, S., Kandula, S., Andrews, C., and Zhou, Y., "Flashback: A lightweight rollback and deterministic replay extension for software debugging," in *Proceedings of the USENIX Annual Technical Conference*, 2004.

[76] Steven, J., Chandra, P., Fleck, B., and Podgurski, A., "jRapture: A capture/replay tool for observation-based testing," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 158–167, 2000.

[77] Tai, K. C., Carver, R. H., and Obaid, E. E., "Debugging concurrent ada programs by deterministic execution," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 45–63, 1991.

[78] Tallam, S., Tian, C., Gupta, R., and Zhang, X., "Enabling tracing of long-running multithreaded programs via dynamic execution reduction," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 207–218, 2007.

[79] Tassey, G., "The economic impacts of inadequate infrastructure for software testing," Tech. Rep. 7007.011, National Institute of Standards and Technology, 2002.

[80] TestPlant Ltd., "Automated software testing for the user interface," 2011. `http://www.testplant.com/products/eggplant_functional_tester`.

[81] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y., "Triage: Diagnosing production run failures at the user's site," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pp. 131–144, 2007.

[82] University of Washington, "Pine© – A Program for Internet News & Email," 2011. http://www.washington.edu/pine.

[83] University of Wisconsin, "The cooperative bug isolation project," 2011. http://www.cs.wisc.edu/cbi/.

[84] Verykios, V. S., Bertino, E., Fovino, I. N., Provenza, L. P., Saygin, Y., and Theodoridis, Y., "State-of-the-art in privacy preserving data mining," *ACM SIGMOD Record*, vol. 33, no. 1, pp. 50–57, 2004.

[85] Vessey, I., "Expertise in debugging computer programs," *International Journal of Man-Machine Studies: A Process Analysis*, vol. 23, no. 5, pp. 459–494, 1985.

[86] Visser, W., Păsăreanu, C. S., and Khurshid, S., "Test input generation with Java PathFinder," *SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[87] VMware, "The Amazing VM Record/Replay Feature in VMware Workstation 6," 2011. http://blogs.vmware.com/sherrod/2007/04/the_amazing_vm_.html.

[88] Wang, R., Wang, X., and Li, Z., "Panalyst: Privacy-aware remote error analysis on commodity software," in *Proceedings of the 17th USENIX Security Symposium*, pp. 291–306, 2008.

[89] Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J., "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 461–470, 2008.

[90] Weiser, M., "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449, 1981.

[91] Weiser, M., "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[92] Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[93] Wilkerson, D. S. and McPeak, S., "Delta," 2011. http://delta.tigris.org/.

[94] Yang, Z., Zhong, S., and Wright, R. N., *Privacy-Preserving Queries on Encrypted Data*, vol. 4189 of *Lecture Notes in Computer Science*, pp. 479–495. Springer Berlin / Heidelberg, 2006.

[95] Young, M., "Perpetual testing," 2011. http://www.cs.purdue.edu/AnnualReports/96/research/perpetual.html.

[96] Zalewski, M., "tmin: Fuzzing test case optimizer," 2011. http://code.google.com/p/tmin/.

[97] Zeller, A., "Yesterday, my program worked. Today, it does not. Why?," in *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 253–267, 1999.

[98] Zeller, A., "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 1–10, 2002.

[99] Zeller, A., "Using Delta Debugging–A short tutorial," 2011. `http://www.st.cs.uni-saarland.de/dd/ddusage.php3`.

[100] Zeller, A. and Hildebrandt, R., "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[101] Zhang, X., Gupta, N., and Gupta, R., "Pruning dynamic slices with confidence," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 169–180, 2006.

[102] Zhang, X., Gupta, R., and Zhang, Y., "Precise dynamic slicing algorithms," in *Proceedings of the 25th International Conference on Software Engineering*, pp. 319–329, 2003.

[103] Zhang, X., Tallam, S., and Gupta, R., "Dynamic slicing long running programs through execution fast forwarding," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 81–91, 2006.

[104] Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A., "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the 23rd International Conference on Machine learning*, pp. 1105–1112, 2006.

[105] Zhong, S., Yang, Z., and Wright, R. N., "Privacy-enhancing k-anonymization of customer data," in *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 139–147, 2005.

VITA

James Alexander Clause was born March 5, 1981 in Nazareth, Pennsylvania, USA. He attended Allegheny College and received the B.S. in Computer Science in 2003. In 2005, he received the M.S. in Computer Science from the University of Pittsburgh and entered the doctoral program at the Georgia Institute of Technology under the advisement of Alessandro Orso. In August 2010, James joined the faculty of the University of Delaware as an Assistant Professor in the Computer and Information Science Department.